SANDIA REPORT

SAND2016-8845
Unlimited Release
Printed September, 2016

Evaluating the Opportunities for Multi-Level Memory – An ASC 2016 L2 Milestone

G.R. Voskuilen, M.P. Frank, S.D. Hammond, and A.F. Rodrigues

Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, 87185

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

SAND2016-8845
Unlimited Release
Printed September, 2016

Evaluating the Opportunities for Multi-Level Memory – An ASC 2016 L2 Milestone

G.R. Voskuilen, M. Frank, S.D. Hammond, and A.F. Rodrigues
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, 87185

The next two Advanced Technology platforms for the ASC program will feature complex memory hierarchies – in the *Trinity* supercomputer being deployed in 2016, Intel's Knights Landing processors will feature 16GB of on-package, high-bandwidth memory, combined with a larger capacity DDR4 memory and in 2018, the *Sierra* machine deployed at Lawrence Livermore National Laboratory will feature powerful compute nodes containing POWER9 processors with large capacity memories and an array of coherent GPU accelerators also with high bandwidth memories.

In this ASC L2 milestone we report on a spectrum of studies investigating the potential performance opportunities of multi-level memory systems which might utilize hardware accelerated caching or, alternatively, entirely software driven management either by an application or allocation/memory aware runtime. As the basis for our exploration we utilize several of the APEX ASC benchmarks currently planned to be used in the ATS-3 *Crossroads* procurement in late 2016.

Our studies investigate issues of system balance, memory size, cache sizes and a number of other key hardware parameters. We show that although a number of benchmark kernels are not bound by the bandwidth of memory in the system and so experience no significant improvement in runtime from higher-bandwidth or on-package memories, other kernels, particularly those relating to sparse linear algebra can experience significant acceleration due to their poor data reuse properties and low ratio of compute operations.

We conclude that multi-level memories provide a very varied picture for the performance of codes on future systems and highlight areas where application programmers and computational scientists may want to focus their efforts.

Acknowledgment

Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This report represents a significant extension to Sandia's Structural Simulation Toolkit that build upon contributions and inputs from the whole SST project team.

Results from the ASC Test Beds draw on work from our diligent system administration teams. We are grateful for their continued support for our work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As new main memory technologies appear on the market, there is a growing push to incorporate them into future architectures. Compared to traditional DDR DRAM, these technologies provide appealing advantages such as increased bandwidth or non-volatility. However, the technologies have significant downsides as well which obviate the ability to completely replace DDR DRAM. High bandwidth stacked DRAM variants such as hybrid memory cubes (HMCs) and high-bandwidth memories (HBMs) require complex manufacturing processes which drive up cost and also can have increased latency compared to traditional DDR DRAM. Non-volatile technologies such as FLASH and PCM have higher latencies than DRAM while technologies like STT-MRAM and PCM have high write energy needs. In addition, all three technologies suffer from wear-out. As such, none emerge as a clear winner compared to DRAM. For these reasons, there is an increased focus on the concept of multi-level memories (MLM), or mixing different memory technologies in a single memory system with the ideal MLM system providing the advantages of all with the disadvantages of none. For example, a system might incorporate a small amount of HMC to support high bandwidth accesses but to reduce cost, use DDR DRAM for capacity. The same system might also add a third non-volatile layer which can selectively back up data from the volatile memories to delay wear-out, while improving reliability and reducing the cost of operations that would typically use the disk (e.g., checkpoints).

In this report, we analyze the effect of next-generation MLM architectures on the performance of key ASC application kernels and algorithms. In particular we look at application performance with a two-level memory system consisting of high-bandwidth HMC-like memory and lower-bandwidth DDR DRAM.

1.1 Multi-Level Memory Systems

1.1.1 The Case For...

Economic Impacts

The primary motivation for a multi-level memory is economic. Replacing DDR main memory with a combination of memory technologies may enable a high bandwidth and and high capacity memory system at low cost. Application analysis (Section 4) indicates that, for many applications, a relatively small percentage of the application's main memory footprint accounts for most cache misses. If this portion were stored in fast, expensive (e.g. 3D stacked) memory and the bulk of data kept in slower, cheaper (e.g. DDR or Flash[17]) devices, it may be possible to realize the "best of both worlds."

Table 1.1 shows a selection of memory technologies which may comprise a future MLM system. Because some of the stacked memory technologies have not started shipping in quantity, prices are only estimates and bandwidth may change as these technologies evolve. However, this table shows the wide set of device criteria and trade-offs.

Table 1.1: Emerging memory characteristics

Memory Technology	Interface, Packaging, Device	Relative Cost Per Bit	Typical Bandwidth / latency
DDR4 DRAM	Parallel multi-package DRAM	1	17 GB/sec (DIMM Module), 35ns
Flash Memory	Multi-package Non-volatile	0.15	2-3 GB/sec, 20-100μs
HMC	Serial 3D Stacked DRAM	3-5?	160-240 GB/sec, 40ns?
HBM	Parallel 3D Stacked DRAM	2-4?	128 GB/sec, 35ns?
Wide IO	Parallel 3D Stacked DRAM	3?	51 GB/sec, 40ns?

Table 1.2: Possible memory system cost breakdown

Memory	Size	Relative Cost/Per Bit	Relative Total Cost
HMC	5%	3.0	0.15
DDR	30%	1.0	0.3
Flash	65%	0.15	0.1
Total			0.54

Analysis

Consider an architecture with a multi-level memory hierarchy based on the technologies from Table 1.1 and optimized for an application like Lulesh (Figure 1.1). The 5% of memory pages that dominate memory accesses could be placed in a small HMC-like memory sized to fit. An additional 30% could be placed in conventional, low-cost DDR, and the "long tail" of infrequently touched memory pages could be placed in non-volatile Flash. Using the cost and bandwidth estimates from Table 1.1, we can estimate the rough cost of the memory system and an upper bound on available "average" bandwidth[1].

Such a memory system could cost about half of a conventional DDR-only memory system (Table 1.2). Since almost half of memory accesses would go to the HMC, the overall "average" bandwidth would be increased (Table 1.3). The majority of remaining accesses would go to DDR, so their performance would be no worse than a conventional DDR-only system. The small portion of accesses that would go to Flash would be slower, but if the latency could be masked (perhaps with intelligent prefetching or application modifications), this may not have a large performance impact. The end result is a memory system that offers significantly increased effective bandwidth and costs about half of a conventional (DDR) system.

[1]The average "effective bandwidth" here is computed as the harmonic mean, weighted by the number of requests, and assuming three independent channels of the listed bandwidth.

Table 1.3: Possible memory system "average" bandwidth

Memory	% of Memory Accesses	Bandwidth
HMC	49.4%	240 GB/s
DDR	25.5%	17
Flash	25.1%	3
Effective Avg.		29.78

1.1.2 The Case Against...

Economics

The biggest potential pitfall for MLM is also economic. Used improperly, MLM has the potential to increase cost and decrease performance. Because some high-performance memory technologies (such as HMC) cost more than commodity DRAM, a memory system which relies too heavily on them will cost more than a conventional memory system. If that memory system is not managed properly, it may not be able to take advantage of the faster memory and will not justify the cost. Similarly, overuse of cheaper non-volatile memory may significantly degrade performance[8].

A related issue is that application diversity implies that no single mix of memory technologoies and sizes will be optimal for all applications. As such, while a given MLM system might be cost-effective for some applications, it may cost more and/or reduce performance for another set of applications. As such, one must carefully consider the applications that will be running on the architecture, and potentially tune those applications to the architecture, to ensure both performance and cost effectiveness.

Management

In addition to economics, a major impediment to successful deployment of MLM technology is its management. Software will need to be modified to place commonly used data in the fast "near" memory and less frequently used data in the "far" slow memory. Alternately, the operating system, runtime, or hardware will have to transparently move data from one memory level to the other by predicting future application requirements.

All of these approaches have substantial costs. Modifying applications or runtimes requires programmer effort, insight, and will probably require a new generation of analysis tools. Adding hardware to manage MLM will incur substantial design costs as well as use additional chip area, increasing cost and possibly power.

Latency

Finally, the simplistic bandwidth analysis presented in Section 1.1.1 only focuses on the bandwidth of new memory technologies. This is insufficient because of the unusual latency characteristics these technologies present. For example, HMC-like memories offer dramatically higher bandwidth and throughput, but their latency is not much better than conventional DDR memory and can even be worse. Non-volatile memories offer lower bandwidth, but also much higher latency (often by orders of magnitude). Additionally, the latency of NV memories is often asymmetric[2]. Considering not only bandwidth but latency in data placement further complicates MLM management. However, if data placement is not properly managed, the potential of MLM can be squandered.

1.2 Three Paths to MLM Management

The memory characteristics of applications of interest to the DOE vary considerably. Figure 1.1 shows the number of post-cache references to different 4K memory pages across three applications. Even this small set exhibits substantial diversity. MiniAero (leftmost graph) shows a small number of well defined regions which are highly accessed. Lulesh shows a large number of regions. RSBench (rightmost) has a very irregular access pattern without well defined regions.

[2]Different read and write latencies.

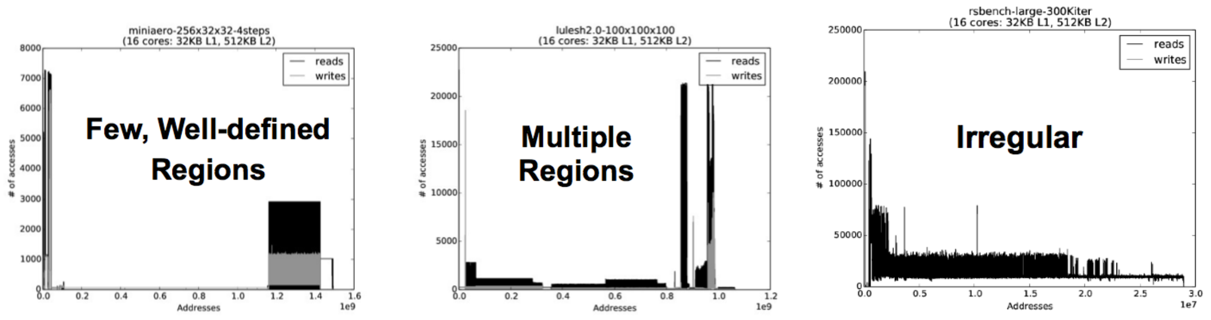


Figure 1.1: Address histograms sorted by address

Because of this application diversity and the trade-offs inherent in different management policies, we believe there is no "One Size Fits All" solution for managing MLM systems. Thus, we explore three paths: Algorithmic, Manual, and Automatic Management.

1.2.1 Algorithmic

Algorithmic management of MLM requires restructuring and rewriting the fundamental algorithms and/or data structures of a code to take advantage of multiple levels of memory. Algorithmic management requires a high level of programmer effort, but has a high potential to improve performance. A disadvantage of the algorithmic approach is that it may require rewrites of applications whenever new levels of memory are added. Additionally, multi-physics applications that contain several algorithms (often contained in separate libraries) may be difficult to adapt to this approach. However, it may be possible for algorithmic restructuring to be combined with other approaches to provide "hints" to the runtime or hardware about data placement.

A detailed analysis of algorithmic restructuring is beyond the scope of this document. However, an application of this approach can be found in a previous paper[1].

1.2.2 Manual

Manual management of MLM requires the programmer to identify blocks of application memory (usually `malloc()` allocations) which should be stored in a given level of memory. This approach is less invasive than algorithmic restructuring, but does require substantial programmer effort to identify and "tag" critical memory regions for placement in fast memory.

An interesting aspect of manual management is that programmer intuition about which memory regions are most critical is often incorrect. Many frequently accessed data structures are are simply moved into the system's SRAM caches[3] and benefit minimally from being placed in faster memory. For example, a thread's stack is accessed very frequently, but produces very few cache misses to main memory. Instead, higher performance is achieved by identifying regions which are accessed frequently, but not *too* frequently. To accomplish this, new program analysis tools will need to be produced.

Chapter 4 covers manual MLM management and introduces MemSieve (Section 4.2), an analysis tool to aid in identifying memory critical regions of an application.

[3]e.g. L1 or L2 cache

1.2.3 Automatic

Automatic management of MLM requires specialized hardware which tracks main memory accesses and coordinates movement of data between different layers of memory. This approach has the benefit of minimizing programmer effort, but does require extra hardware. This hardware requires tables to track accesses, DMA engines to transfer between layers, and a TLB-like structure to determine in which memory level a given address resides. Together, these structures could be power- and area-hungry, increasing cost.

An interesting finding of our analysis of automatic management policies is that they differ from conventional caching policies. Specifically, most cache policies focus on *replacement* – deciding what to remove from a cache when you need to add data to it. In contrast, MLM is much more sensitive to *addition* policies – deciding if it is worth bringing something into the "fast" memory when it is accessed. This is because the latency of "fast" memory is so similar to that of conventional DDR DRAM.

Chapter 5 explores automatic management.

1.3 Roadmap

Chapter 2 discusses the methodology used in experiments in this report. Chapter 3 discusses a design space exploration of HMC different configurations. Chapter 4 covers manual MLM management approaches. Chapter 5 covers automatic MLM management approaches. Finally, Chapter 6 presents conclusions and future work.

Chapter 2

Methodology

In this chapter we present our experimental methodology. As mentioned in the previous chapter, we analyze the performance of ASC codes at the node level on a two-level memory system with near HMC and far DDR. Because the hardware and software infrastructures do not yet exist for the architectures and management strategies we are evaluating, we primarily employ simulation. However, we do use real hardware when available to validate simulation results. In Section 2.1 we present the applications studied and in Section 2.2 we describe our simulation infrastructure including simulation validation studies and how we enabled simulation at scale. Finally, we present the two architectures used throughout our studies in Section 2.2.2.

2.1 Applications

A major call in this milestone was to analyze the impact of MLM on key kernels and algorithms found in ASC codes. To that end, we selected four mini-apps from the APEX benchmark suite. This suite is designed to be used by vendors and labs to evaluate proposed systems for the 2020 Crossroads/NERSC procurement. As such, the mini-apps cover a wide variety of algorithms used across multiple labs. By studying these applications, we hope to gain insight that can be used to steer procurement decisions, enable meaningful interaction with vendors about future architectures, as well as assist application developers in effectively using systems with MLM. For this milestone, we selected four applications, MiniPIC, PENNANT, HPCG, and SNAP. Additionally, to increase confidence that our results translated to real systems, we opted to simulate these applications at larger scales than have previously been attempted. In this study each benchmark is sized to yield a data footprint of 1GB to 8GB, depending on the particular experiment. Simulating at this scale required a number of simulator improvements (Section 2.2) as well as a sampling methodology which we will discuss in the following subsection. The input parameters and memory usage for each application is shown in Table 2.1. A brief description of each mini-app is provided below.

MiniPIC [2] uses the particle-in-cell (PIC) method to solve the discrete Boltzman equation in an electrostatic field in an arbitrary domain with reflective walls. It solves this equation using the following algorithm on an unstructured mesh. Particles are weighted to the grid as a charge density and Poisson's equation is solved. The electric field is then calculated and weighted to the particle locations. Finally, the particles are accelerated and moved to new locations. As particles interact with each other and walls, new particles may be created. This process is repeated in each time step. MiniPIC uses the Trilinos Tpetra library for matrix and vector operations. For scalability at a high level, MiniPIC spreads the field over MPI ranks; once a particle cannot move any farther in its rank, it moves to a neighboring one. For node-level parallelism, MiniPIC employs threading via the Trilinos Kokkos package. Kokkos enables performance portability by allowing users to select architecture-specific threading models (e.g., CUDA or OpenMP). In this study we use the OpenMP backend and do not use MPI. MiniPIC was developed at Sandia National Laboratories.

PENNANT [5] is a hydrodynamics algorithm which uses Lagrangian and staggered-grid methods. The algorithm is implemented over a 2D unstructured finite-volume mesh. For parallelism, PENNANT does geometric domain decomposition across MPI ranks and, at the node level, employs OpenMP threads to divide the local mesh into (almost) independent computational chunks. PENNANT was developed by Los Alamos National Laboratory. During our analysis we identified that PENNANT has a significant number of

Table 2.1: Application parameters

Application	Version	Input	Memory	Parameters
MiniPIC	APEX release on 2/26/16	big	8G	Nx=Ny=Nz=7; `--dt=0.5 --tfinal=2.0 --nparts=8000`
MiniPIC	APEX release on 2/26/16	small	1.8G	Nx=Ny=Nz=4; `--dt=0.5 --tfinal=2.0 --nparts=8000`
PENNANT	0.9	big	8G	leblancbigx6 (same as default leblancbigx4 with 'meshparams 960 8640 1.0 9.0' and 'dtinit 1.67e-4'
PENNANT	0.9	small	1G	leblancbigx6 (same as default leblancbigx4 with 'meshparams 320 2880 1.0 9.0' and 'dtinit 5.e-4'
HPCG	3.0	big	8G	nx=272, ny=272, nz=136
HPCG	3.0	medium	4G	nx=192, ny=192, nz=136
HPCG	3.0	small	1G	nx=112, ny=112, nz=112
SNAP	1.06	big	8G	APEX 'in_s' input modified with npey=1, npez=1, nx=128, ny=16, nz=20, ichunk=16
SNAP	1.06	small	1G	APEX 'in_s' input modified with npey=1, npez=1, nx=32, ny=12, nz=16, ichunk=16
MiniFE	2.0	small	11M	`-nx 30 -ny 30 -nz 30`
Lammps	2/16/16 release	small	25M	`-i in.eam -sf omp`
Lulesh	2.0.3	small	10M	`-s 20 -i 10 -r 11`
CoMD	1.1	small	19.5M	`--nx 25 --ny 20 --nz 20 --nSteps 5`

small allocations (shown later in this report). Although we have not modified the code for the remainder of this study, in part to ensure we analyze the *existing* code, the use of an optimized allocator or memory pool scheme could significantly improve the performance of PENNANT on some architectures.

HPCG [4] is an open-source benchmark that was developed as an alternative method to High Performance LINPACK (HPL) for ranking HPC systems. HPCG does a fixed number of conjugate gradient iterations using a simple multigrid preconditioner. Like the previous applications, HPCG uses both MPI and OpenMP parallelism. In particular, OpenMP parallel for loops are used for many of the vector and matrix operations. Additionally the SpMV algorithm is threaded over matrix rows.

SNAP [18] is representative of discrete ordinate neutral particle transport applications. Although it implements no 'real' physics, SNAP was developed to mimic the computational workload, data layout, memory usage, and communication patterns found in real applications. SNAP executes a number of timesteps which are structured in a two level nested loop. At each timestep, the outer loop executes until either a set number of iterations have been executed or convergence is reached. Likewise, within each outer loop iteration, an inner loop is executed until a set number of inner iterations have been executed or convergence is reached. The majority of computation occurs within inner loop iterations, however outer iterations do compute an outer source sum as well as check for convergence. SNAP uses both MPI and OpenMP for parallelism. At the node-level, OpenMP parallel loops are used for the convergence testing functions, to sum inner and outer sources, SNAP was developed by Los Alamos National Laboratory.

In addition to the above mini-apps, we also used a number of other applications for early design space exploration studies and some validation work. Unlike the previous applications, we do not execute these applications at scale, enabling us to quickly prune the design space. For validation in particular, we used the STREAM benchmark which exercises system bandwidth. For early analysis studies, we used Lammps[13, 14], a molecular dynamics application; MiniFE[6], an unstructured implicit finite element code; Lulesh[10], a hydrodynamics code; and CoMD[6], a molecular dynamics application. Each application was executed with 16 threads. Additional parameters and the memory footprint is listed in Table 2.1. These latter mini-apps were selected because they exhibit a diverse set of main memory access patterns[8]. MiniFE has a small number of memory regions (contiguous pages) with similar access characteristics and a low Gini coefficient, indicating a relatively equal number of accesses per page. Lulesh has multiple different regions and a high Gini coefficient, indicating a small set of pages that are accessed disproportionately. CoMD and Lammps have more irregular access patterns.

2.1.1 Methodology for Simulating at Scale

Although we made numerous scalability improvements to the simulator (discussed in Section 2.2), simulating the APEX mini-apps in full at the scales shown in Table 2.1 would require weeks to months or more. Thus, to enable simulation at those scales, we adopted a sampling methodology. For each application we sampled one to two iterations of the main computational loop at each of the beginning, middle, and end of the application's execution. For SNAP, we considered the inner loop to be the main loop. For HPCG, we consider iterations of the main CG loop, including the preconditioner. For PENNANT, we look at iterations of the main event loop in the Driver class. Finally, in MiniPIC, an iteration is a timestep. Specific parameters for this sampling are shown in Table 2.2 for the different problem sizes shown earlier. For HPCG, the sampling is the same for all problem sizes. While this method allowed us to simulate PENNANT, HPCG, and SNAP in a reasonable time frame, additional sampling was needed for MiniPIC. As described above, during each timestep, MiniPIC does a number of operations on the particles. Therefore we further divided a timestep into three parts. Particle charge weighting and solve fall in the first part, *charge*. The second part, *field* includes the electric field calculation and weighting. Finally, the third part, *move*, contains the particle acceleration and move functions. Even so, we were unable to simulate *field* and *move* in full. Instead we measured over the calculations for the first N particles, as shown in Table 2.2. We used the same 'N' values in the begin, middle, and end samples so we show them only for the beginning sample in the table.

Table 2.2: Application samples

Application	Begin (big)	Middle (big)	End (big)	Begin (small)	Middle (small)	End (small)
MiniPIC	t=0.0; *field* N=10M; *move* N=5000	t=0.5	t=1.5	t=0.0; *field* N=1.5M; *move* N=300K	t=0.5	t=1.5
PENNANT	cycles 0-2	5000-5002	11100-11102	0-2	3700-3702	7470-7472
HPCG	iterations 0-1	24-25	49-50	0	25	50
SNAP	inner=1, outer=1, timestep=1	i=4, o=4, t=3	i=3, o=5, t=8	i=1, o=1, t=1	i=4, o=4, t=3	i=3, o=5, t=8

Without simulating more of the application, we cannot be certain that an averaging the samples accurately reflects the average application performance. Therefore in our studies we will combine samples only when they are nearly identical, otherwise results for each each sample will be given. In general, HPCG's and PENNANT's samples were very similar. In SNAP, the beginning sample differed from the latter two samples which were similar. Finally, in MiniPIC, the samples across timesteps were similar, but the samples within timesteps tended to differ.

2.2 Simulation

Having defined the applications, we now discuss our simulation infrastructure. We used the Structural Simulation Toolkit (SST) [15] version 6.0 for this study. SST is a parallel discrete event simulation engine developed at Sandia. In addition to providing a simulation engine, SST also includes a number of architectural *component* models (e.g., processors, networks, memories) that can be connected to form different simulated

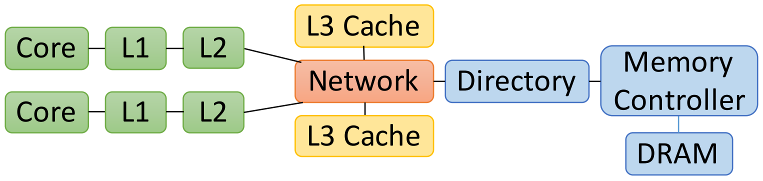


Figure 2.1: Generic SST configuration

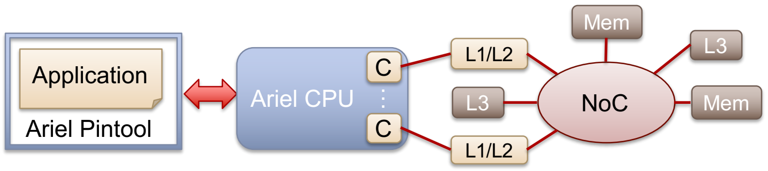


Figure 2.2: An SST configuration with Ariel, MemHierarchy, and Merlin

system architectures. An example of how an architecture might be constructed from individual components is shown in Figure 2.1. We describe the components used in this study below.

We used the *Ariel* processor model for our multicore processor. Ariel consists of two parts that interact via a shared memory region. The first is a PIN tool that utilizes Intel's PIN framework [11] to instrument a native binary. As the binary runs on the host processor, Ariel's PIN tool intercepts memory instructions and passes them to the second part, the simulated processor. The simulated processor implements multiple cores, one per thread. It also models a page table manager which manages memory allocation and deallocation as well as virtual-to-physical address translation. The queue between the simulated processor and the PIN-instrumented binary enables the simulation to stall the binary execution when it runs too far ahead of the simulation. As such, Ariel allows the use of native binaries, is fairly lightweight, and accurately models memory system traffic although it does not model the full execution pipeline and functional units. To approximate computation, Ariel uses no-ops when there are no memory instructions to execute (i.e., the application is tied up in computation). Figure 2.2 illustrates how Ariel fits into the larger simulation.

MemHierarchy is a collection of memory component models including caches, buses, directory controllers, and memory controllers. MemHierarchy is a cycle-level model that provides highly flexible cache topologies with directory-based coherence (MSI and MESI). In addition to these components, MemHierarchy implements memory controller interfaces, called memory backends, to a number of memory simulators. To model the DDR in this study, we used the DRAMSim simulator developed at University of Maryland. To model HMC, we used Texas Tech's Goblin HMC simulator and VaultSim, a generic vaulted memory component provided by SST.

For the on-chip network (NoC) model, we used SST's Merlin component. Merlin is an flexible, cycle-level network simulator that can be used to model large scale intra-node and off-chip networks as well as the smaller on-chip networks needed for our single node studies. Merlin can implement arbitrary network topologies and models traffic at the flit level.

In the following section we describe the scalability, performance, and functional improvements that were necessary for simulating MLM at the scales presented in the previous section. We then discuss the two architectures we considered along with their simulation parameters. Finally we present the results of some validation studies.

2.2.1 Simulation Improvements

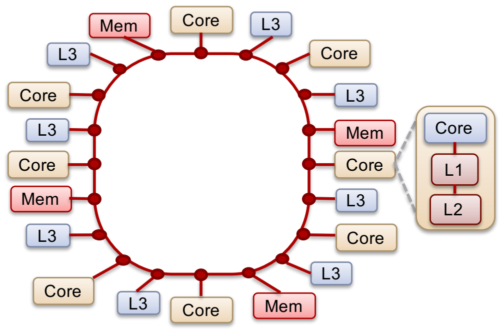
To enable scalable simulation, we made a number of improvements to the Ariel and MemHierarchy components. To reduce execution overhead, we re-wrote the MemHierarchy caches, directory, and memory NIC to be primarily event driven instead of clock driven. While clocks are still used, they are deactivated unless events need to be handled. This change reduced run time by approximately 30% for the STREAM benchmark. The second major improvement involved streamlining and reorganizing code in both MemHierarchy and Ariel to eliminate unnecessary compute. We found that these changes reduced runtime by a further 5-15% (depending on the change). Additionally, to reduce the memory overhead, we added the option to simulate without backing the memory system (i.e., tracking memory values at the memory). For simulations with Ariel where simulated data values do not affect the application execution, we can reduce the footprint from the size of the simulated memories (e.g., 10s of GBs) to one GB or less. All of these improvements were incorporated into the SST 6.0 release.

In response to the validation studies described next, we also made some simulated performance improvements. This included support for fine-grained address striping across memories, eliminating a bottleneck at the memory NIC which handles address to destination translation, and improved control over throughput at the memory controller. We also fixed a number of performance bugs that had not been seen at smaller simulation scales. One significant bug was that the cache set index calculation at cache arrays was not distributed-cache aware leading to unused cache sets. In turn, depending on the number of cache sets and the number of distributed cache slices in a system, the distributed cache capacity was effectively lowered. Fixing these bugs and adding the additional capabilities allowed us to realistically model real architectures.

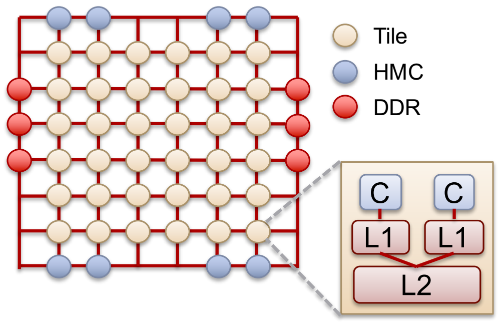
Finally, to support multi-level memory, we added a number of capabilities in Ariel to allow applications to manage the memory. We extended the Ariel API which provides applications with 'hooks' to control and interact with the simulation (e.g., start the simulation, trigger a statistics dump, get the simulated time, etc.) with operations for allocating memory in specific memory pools. While multi-level memory refers to separate memory types as "levels", Ariel operates with the concept of memory "pools". Each pool can be targeted at a separate memory and the new API calls allow the application to allocate and free memory directly in particular pools as well as to flag that upcoming mallocs should be allocated to a specific pool.

2.2.2 Architectures

As mentioned in the introduction, to determine whether applications responded similarly to the presence of multi-level memory independent of the architecture, we studied two architectures with completely different network topologies, processor capabilities, and memory hierarchies. Each architecture reflects a prevailing design strategy found in real hardware. The first, *heavyweight* architecture, has a few (i.e., 8-16) large, powerful cores connected to a deep cache hierarchy. Many traditional architectures, such as the Intel Xeon, IBM Power, and AMD Opteron fall in this category. The second architecture, *lightweight* features many less powerful cores with a shallower cache hierarchy. An example of this kind of architecture would be the Intel Xeon Phi. Heavyweight architectures tend to maximize single-thread performance while lightweight architectures are designed for high throughput. The specific architectures studied here are shown in Figure 2.3. The simulation parameters are given in Table 2.3. Note that the purpose of studying these two architectures is not to determine which of the two paths is better. Rather we seek to determine how different architectures affect applications' memory access patterns and ability to use multi-level memory. Accordingly, we do not attempt to equalize the available memory bandwidth in the systems or otherwise single out the effect of particular architectural features. In terms of real hardware, the lightweight model resembles an Intel Knights Landing (Xeon Phi) while the heavyweight model is similar to the Intel Sandy Bridge (Xeon) architecture.



(a) Heavyweight architecture



(b) Lightweight architecture

Figure 2.3: Architectures studied

Table 2.3: Simulation parameters

Architecture	Component	Parameters
Heavyweight	Processor	8 cores, 2.66GHz, 3 reqs/cycle
	Caches	Private 32KB L1 (8-way assoc, 4 cycles), Private 256KB L2 (8-way assoc, 6 cycles), Shared distributed L3 with 8 2MB slices (16-way assoc, 12 cycles), MESI
	Memory	2 HMC @ 160GB/s, 2 DDR @ 20GB/s, capacities vary by experiment
	Network	Ring (merlin.torus), 96GB/s, 300ps link latency
Lightweight	Processor	72 cores, 1.4GHz, 2 reqs/cycle
	Caches	Private 32KB L1 (8-way assoc, 2 cycles), 512KB L2 cache shared between 2 cores (16-way assoc, 6 cycles), MESI
	Memory	8 HMC @ 160GB/s, 6 DDR @ 20GB/s, capacities vary by experiment
	Network	8x8 mesh, 57GB/s, 50ps link latency

2.2.3 Validation

To validate our simulation models, we performed two studies. The first study validated that the Goblin HMC simulator's bandwidth characteristics matched that of real HMC hardware. The second validated that for the STREAM benchmark we could attain similar bandwidth on a simulated Sandy Bridge architecture and a real Sandy Bridge chip. Additionally, near the end of this study, early Knight's Landing (Xeon Phi) hardware became available. While we do not rigorously validate our lightweight architecture against it, in part because the software environment for the Xeon Phi (Knights Landing) is still rapidly changing, we will present some comparisons throughout the study.

To validate the Goblin HMC simulation model, we ran a number of access patterns through our Pico EX800 HMC testbed. This testbed features a 2GB HMC connected to four Stratix V FPGAs (one per link). The access patterns were chosen to exercise a variety of contention patterns. Each pattern generates random accesses from a specific link to a specific portion of the HMC (e.g., quad, vault, or bank). Figure 2.4 illustrates the patterns. In the top row, links access the colored region in their local quad. In the bottom row, all links access the same colored region. The normalized bandwidth measurements from these access patterns are shown in Figure 2.5 in red. Measurements are normalized to the bandwidth of accessing random addresses in the link's local quad ("own quad"). The next bar shows the original GoblinHMC simulator. While the simulated bandwidth matches the same trend as the measured bandwidth for access patterns at the quad and bank level, the vault patterns ('own vault' and 'same vault') differ significantly from the measured pattern. In response to this validation study, a change was made to the simulated vault controller's contention management which resulted in the tan bar in the figure. Although the absolute numbers still vary in a few cases, the hardware and simulated trends now match.

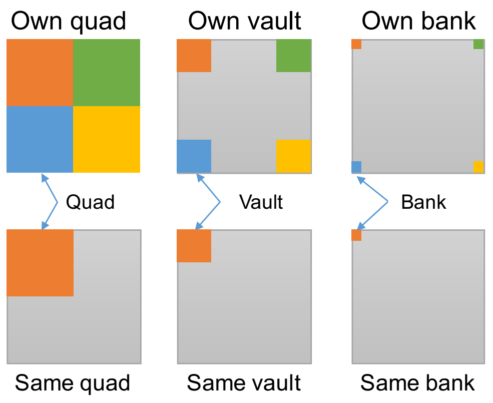


Figure 2.4: Studied access patterns: links access areas of their local quads (top) and links access the same area in a single quad (bottom)

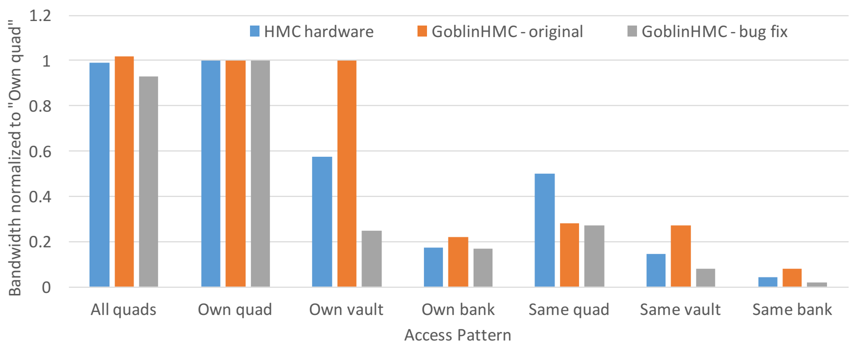


Figure 2.5: GoblinHMC validation study

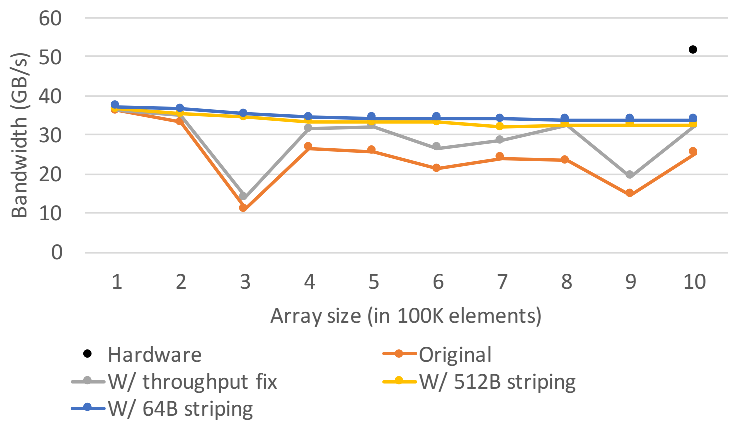


Figure 2.6: Sandy Bridge socket validation study

The second validation study we did was to verify that the simulated Sandy Bridge socket described in the previous section matches the STREAM performance of real hardware. This validation implies that we are correctly modeling system bandwidth. For this study we ran the STREAM benchmark at a variety of scales (array sizes) on the simulated system and compared to real hardware. Figure 2.6 shows the expected (measured hardware) performance in black and the performance of the original model running in SST 5.1, in orange. Array scale in hundreds of thousands of elements is shown along the X axis, bandwidth is shown on the Y-axis. While the simulation achieves reasonable bandwidth for a small 100K array, its performance quickly drops off. Analysis revealed that there was significant queuing occurring at the output of the directory controller NIC. This occurred because aside from the network bandwidth and core's limits on the number of requests issued per cycle, the NIC was (incorrectly) the only limiter of throughput in the system. Adding limits on throughput to the directory and memory controllers and allowing the NIC's throughput to match the network bandwidth resulted in the gray line in Figure 2.6. While performance is reasonable at some array scales, at certain scales (namely 300K, 600K, and 900K), STREAM performed very poorly compared to the real hardware. Because the "bad" scales were all multiples of 300K we suspected that the array size was interacting pathologically with some addressing in the memory system. By default, MemHierarchy stripes contiguous 4KB chunks of memory across the memory controllers in the system. Reducing the granularity from 4KB to 512B and finally 64B (i.e., a single cache line) as shown in the yellow and darker blue colors respectively, eliminated the pathological cases. Still the performance of the simulated system is a somewhat worse than the real system. This deficit stems from the hardware using streaming stores to reduce pressure on the memory system, something that SST does not yet support. While we did not implement them for this study, based on these results, we plan to incorporate streaming store support into future SST releases.

With these validation studies and their resulting simulation improvements, we are confident that our simulated models will yield the same trends we would see if we had hardware available. To further build confidence in our results, where possible throughout the next chapters we will also compare to measurements on an early version of the Knight's Landing (Xeon Phi). This hardware has both high-bandwidth memory and DDR and resembles the 'lightweight' architecture from the preceding section. However because the hardware is still changing, we do not attempt a rigorous validation study.

Chapter 3

Design Space Exploration

Before analyzing multi-level memory management in the following sections, we look at the design space for the HMC memory. We begin by analyzing the potential performance improvements when moving from traditional DDR DRAM to the higher bandwidth HMC.

3.1 Potential Performance with HMC

To set a baseline for what is possible with the higher bandwidth of the HMC, we consider architectures with enough HMC capacity so that the applications can execute fully out of the HMC. The following two figures show the potential performance possible if the entire memory system were to be HMC instead of DDR or DDR and HMC together. The heavyweight architecture is shown in Figure 3.1 and the lightweight in Figure 3.2. Performance (Y-axis) is normalized to the performance of the same architecture with all memory as DDR. Applications are shown on the Y-axis. As described in Section 2.1.1, we show a single bar for HPCG and PENNANT as their samples are nearly identical, but split out the samples for the other applications. For MiniPIC, the samples within an iteration differ but the same sample across iterations does not differ (i.e., begin-charge is similar to middle-charge but not begin-field). Therefore we show a separate bar for each of the samples within an iteration, *charge*, *field* and *move*. For SNAP, the begin sample differed from the middle and end samples and so we show two bars, one for begin (SNAP p0) and one for the middle and end (SNAP p1/2). Finally, recall that because the bandwidth difference between DDR and HMC is 8X, that is the maximum expected performance gain.

Looking at the heavyweight architecture, three of the four applications (SNAP, HPCG, and PENNANT) benefit from the increased memory bandwidth. HPCG and PENNANT perform especially well with 4.8X and 6.4X speedups respectively. SNAP benefits to a lesser extent with a 1.7X speedup. This indicates that both HPCG and PENNANT are highly memory-bandwidth bound and SNAP is less so. For MiniPIC, the *charge* and *move* samples show no performance improvement. The *field* samples has a 1.3X speedup. This

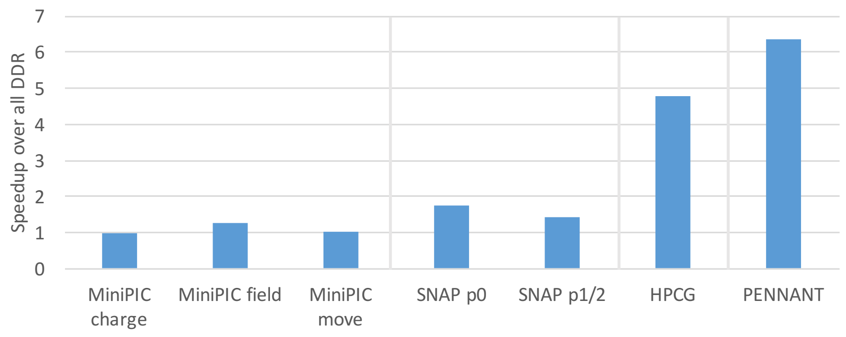


Figure 3.1: All-HMC performance for the heavyweight architecture

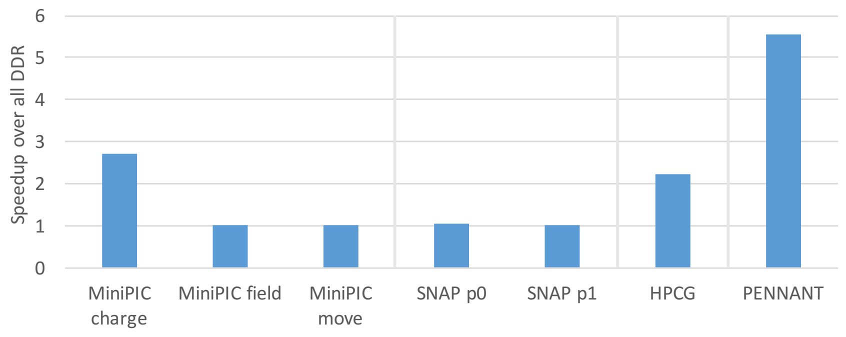


Figure 3.2: All-HMC performance for the lightweight architecture

occurs because on the HMC model, MiniPIC varies for this sample. In some executions it achieves just a 1.0X to 1.05X improvement, while in others it achieves nearly a 1.7X improvement. We have not confirmed the reason for this variance but believe it is tied to the order that particles are processed (i.e., inserted onto the particle list). An algorithmic change to ensure a "good" ordering might ensure that this phase of execution always performs well. However, the improvement may not benefit the application as a whole as the *move* phase dominates execution time.

The lightweight architecture in Figure 3.2 shows a similar trend to the heavyweight. MiniPIC has no improvement - while the *charge* sample does get a speedup, *charge* accounts for a tiny fraction of the application runtime. In contrast to the heavyweight where SNAP improved moderately, replacing DDR with HMC in the lightweight architecture does not affect SNAP's performance. However, both HPCG and PENNANT show large performance gains, as they did for the heavyweight architecture. Overall, these results indicate that gains in one architecture translate to gains in the other, although the magnitudes of the performance gains vary because the architectures are so different.

3.2 Validating the Lightweight Model

As mentioned previously, we do not fully validate the lightweight model due to our changing Knight's Landing hardware, but we do compare to the trends on that architecture. We note that these trends represent our current early version of the hardware and therefore may still change. In Figure 3.3, we show the simulated performance on the Y-axis for each application running on the lightweight architecture with all-HMC normalized to all-DDR. The second data series shows the performance for the same applications running on Knight's Landing hardware. These bars show the performance of running entirely out of the HBM compared to running only out of the DDR. We observe that the overall trends match between the simulated and hardware cases. Essentially, for applications that the simulation has predicted an improvement, the hardware shows an improvement. The magnitudes of the gains vary. This is due to some differences between the hardware and the model, including a slightly different thread count - 72 simulated threads compared to 64 on the hardware, simulated samples compared to running the full application, and the fact that the hardware is yet evolving.

3.3 Latency and Bandwidth Sensitivity

To look at the sensitivity of these workloads to latency and bandwidth, we compared three variations of the HMC to the baseline HMC. For latency, we experimented with increased latency as this is the expected trend for memory. As additional capability such as processing-in-memory and deeper stacks are added, we

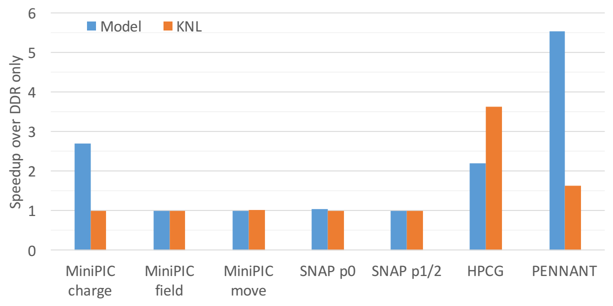


Figure 3.3: All-HMC performance for the simulated lightweight architecture compared to a Knight's Landing hardware testbed

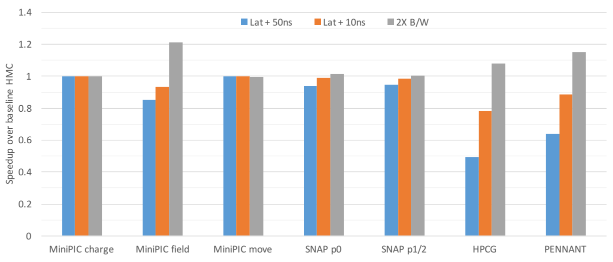


Figure 3.4: Bandwidth & latency sensitivity

expect the HMC latency to increase slightly. To model this, we compared the baseline HMC to one with a 10ns increase in latency (15%). To look at the effect of a significantly higher latency memory, as might be the case with future, non-DRAM based memory technologies, we further compared to an HMC that had a 50ns increase in latency (75%). For bandwidth, because bandwidth is expected to double in the future, we looked at the performance change when moving from a 160GB/s (baseline) HMC to a 320GB/s HMC. To support the increased bandwidth, in this configuration we also increased network bandwidth and the memory controller throughput. Because the heavy and lightweight architectures have similar trends, we show results only for the heavyweight architecture.

The performance for these three variants normalized to the performance of the baseline HMC is shown in Figure 3.4 on the Y-axis (higher is better). Applications are shown along the X-axis. The two latency variants are shown by the *Lat + 10ns* and *Lat + 50ns* bars. The bandwidth increase is shown by *2X B/W*. For the applications that demonstrated a smaller potential improvement with HMC, namely the MiniPIC *charge* and *move* samples, as well as SNAP, there is little reaction to changes in latency or bandwidth. However, both HPCG and SNAP show a significant reduction in performance with increasing latency. To a smaller extent, MiniPIC *field* also shows a performance reduction. In contrast, these applications all improve with the doubled bandwidth. Because the bandwidth increased by just 2X, the improvement is not as great as that of going from DDR to HMC where bandwidth increased by 8X. Still, it is clear that for some applications, increasing bandwidth will continue to improve performance. Overall, we conclude that these applications can benefit from higher bandwidth, but not at the cost of higher latency.

Chapter 4

Manual Management

4.1 Software Approaches & Trade-offs

In this section we evaluate methods for managing multi-level memory (MLM) in software. As mentioned in Chapter 1, software management is attractive because it has the potential to be very efficient (albeit with significant manual effort) and because it does not rely on hardware vendors for its implementation. We describe three major strategies for such management. In algorithmic, an application writer employs different data structures and algorithms to align the application's memory behavior to the target memory technology. For example, the writer might restructure the code so that a section of code can operate completely out of high-bandwidth memory and make use of the increased memory bandwidth. Because of the demand on programmers, as well as the need for algorithm-specific optimizations, we don't explore this concept deeply here. Still, in the following analysis, we will make note of areas where code changes might complement the other management techniques. A second strategy is to utilize OS or runtime managed MLM. In this technique, the OS determines where to allocate memory and may even migrate that memory between memory levels if needed. A simple OS approach might be application-unaware while more complex managers might monitor the application and memory performance to inform the management. Finally, the third management strategy, manual, requires that the application manage its own memory. Manual management generally requires code changes to specify where data should be allocated, as well as to migrate data between memory levels if necessary. Note that OS/runtime management and application management are not mutually exclusive. One can imagine systems where some memory is handled by each, as well as systems where the OS manages the MLM in consultation with the application.

We focus on these last two, OS and application management. In addition to these primary methods, we evaluate whether these decisions can be made statically for a run versus dynamically throughout the execution of the application. In static allocation, the OS or application writer would make its allocation decisions once and they would remain for the entirety of the execution. The advantage of static is its simplicity - no re-evaluating mapping decisions and no re-mapping and its accompanying latency and bandwidth overheads. However static allocation has the downside of likely not being sufficient for applications which have distinct phases where different memory regions are touched. The second variable is whether allocation is done at a page or malloc granularity. This variable does not apply to application-managed MLM as applications do not use the concept of pages and therefore can only operate at the malloc level. For the OS however, there are trade-offs. In general, large pages and mallocs have a higher probability of containing both bandwidth-bound and bandwidth-insensitive regions. Putting either in an HMC wastes space that could otherwise be given to only bandwidth-bound pages or allocations. Further, with very large allocations, one may not be able to pack mallocs into the HMC and use the entire available memory.

In the following subsection we look deeper at the programmer effort, the expected performance, and the potential overhead involved in both OS/runtime-managed MLM and manual/programmer managed MLM. In Section 4.2 we present *MemSieve*, an analysis tool we developed to analyze an application's memory use. Finally, in Section 4.3 we compare the performance of the manual approaches, including static and dynamic allocation and malloc versus page-based allocation granularity.

4.1.1 Trade-offs

Application- and OS-managed MLM trade-off programmer effort for potential performance. We summarize the key advantages and disadvantages of each management method (OS and application) in Table 4.1. OS-management has the advantage of requiring little to no effort from the programmer. For legacy applications this is attractive. On the other hand, the OS has no knowledge of the program behavior and so must either blindly guess where to allocate memory or must use potentially high-overhead runtime profiling to intelligently steer its decisions. Such profiling is complicated by the OS's lack of visibility into memory usage. Unless there is a TLB-miss the OS is not involved in memory accesses. Further, even if the OS were monitoring accesses, the OS does not differentiate between accesses that hit in a cache versus those that go all the way out to memory. This distinction is important as many frequently-accessed regions of memory end up being cache resident and so do not use main memory bandwidth.

Table 4.1: Trade-offs in management approaches

OS/Runtime	Application
+ Able to capture allocations not under application control	+ Knowledge of program behavior for better allocation
+ No intervention from programmer required	+ Application analysis to determine memory behavior can be done offline
- No program knowledge for smart allocation	- No standard way to manage allocation within local arrays or library calls
- Intelligent allocation requires programmer assistance or runtime profiling	- Pervasive code changes
- Increased page-table complexity	- Offline analysis may be difficult if memory use varies widely with input
- Potentially expensive re-mapping to support dynamic	- Potentially need to repeat offline analysis every time there's a new architecture

In contrast to OS-management, application-managed MLM also incurs overhead but most is offline. The application writer must spend time profiling the application to determine how it uses memory, as memory use is not always obvious (e.g., many frequently used allocations end up in cache). The writer must then edit the application to explicitly manage memory, including moving and copying allocations if dynamic migration is needed.

If the application requires dynamic migration, further support must be added to the application to move allocations between memories throughout execution. Both OS and application-managed MLM incur overhead to determine how an application uses memory. However, while the OS would incur runtime overhead to determine application behavior, the application incurs overhead offline to analyze its memory usage. Because the application-managed analysis is offline, applications whose memory use is very sensitive to the input may not perform as well with offline analysis as compared to online.

OS management of memory is easier for the programmer and able to capture allocations that are part of startup or libraries (for which the programmer may have no control). An example of this approach is the use of the `numactl` utility on multi-socket or multi-memory-pool platforms such as Xeon, POWER or Knights Landing. These approaches can help to limit the impact to application source code and still provide the ability to target higher bandwidth hardware where available. The downside of such approaches is operating system complexity, possible conflicts between the application and libraries, and the lack of knowledge the operating system has of how data allocations are used resulting in the potential for lost performance.

Having discussed the trade-offs with different approaches, we now present the approaches studied in this chapter.

4.1.2 Options Explored

We examine four policies for MLM management which range in expected performance from low to high. On the low side, we look at statically and greedily allocating pages ("Policy 1") and mallocs ("Policy 2") to HMC, falling back to DDR when the HMC is full. Such policies fall under OS management and would be the very minimum that the OS could do. On the high side of expected performance we look at application-managed manual allocation, both statically ("Policy 3") and dynamically ("Policy 4"). Comparing static and dynamic allocation will tell us when an application might benefit from dynamic or whether static is sufficient. Further, the performance gap between the static/greedy approach and the manual approaches will indicate how much work would be needed on the part of an OS or runtime to reach the performance of manual allocation. Such work might involve prediction, programmer "hints", and/or runtime profiling. A small gap indicates very little work is needed – simple prediction may suffice, whereas a large gap indicates that both the OS and application programmer may need to be involved in the allocation (i.e., a hybrid approach).

Before evaluating each policy, we analyze the memory behavior of our applications. By studying this behavior we are able not only to inform manual allocation but to predict whether an application will be amenable to a particular policy. The following section presents this analysis.

4.2 MemSieve: A tool for profiling application memory behavior

Analyzing the memory behavior of applications and how that behavior changes with application input and architecture enables us to predict how applications will respond to different memory architectures and management strategies. Specifically, for a multi-level memory system containing high-bandwidth HMC and conventional DDR, we are interested in regions of memory which disproportionately use memory bandwidth. These regions are likely to benefit from being placed in HMC. To measure memory bandwidth use we define the metric, *access density*, to be the number of accesses to a region divided by the size of the region. We hypothesize that the denser a region, the more likely it is to benefit from allocation in HMC. The particular regions we consider here are each allocation (malloc) made by an application as they are the finest granularity that can be easily captured by application profiling and mapped back to application memory objects. However, one could easily extend the concept of *access density* to larger or smaller regions.

To capture *access density* we developed an SST-based tool, MemSieve, that correlates main memory accesses (i.e., last-level cache misses) to application allocations (mallocs). It does so without simulating the full cache and memory hierarchy, making it faster and more scalable than using detailed simulation to collect the same information. As our eventual goal is to apply MemSieve to full applications, we consider this capability critical. In our experiments, MemSieve achieves at least a 2.5X speedup over full simulation, yet yields similar main memory access statistics (miss rates, read and write counts). Because MemSieve is a simulation tool, we are able to vary the architecture it models (e.g., heavy vs lightweight) to determine how architecture affects memory use. In contrast, vendor tools tend to be tied to a particular architecture and so cannot be used for forward-looking analysis.

4.2.1 The MemSieve Tool

MemSieve consists of two parts, a PIN tool and processor model (we use Ariel) which records memory allocations and issues memory requests, and MemSieve itself, which models essentially, an un-timed last-level cache. A block diagram is shown in Figure 4.1. Although we show a single MemSieve in this diagram, we can use multiple MemSieves in a system to model multiple semi-shared last-level caches, as in the lightweight architecture.

Because we are modeling very little detail about the memory system other than its last-level cache

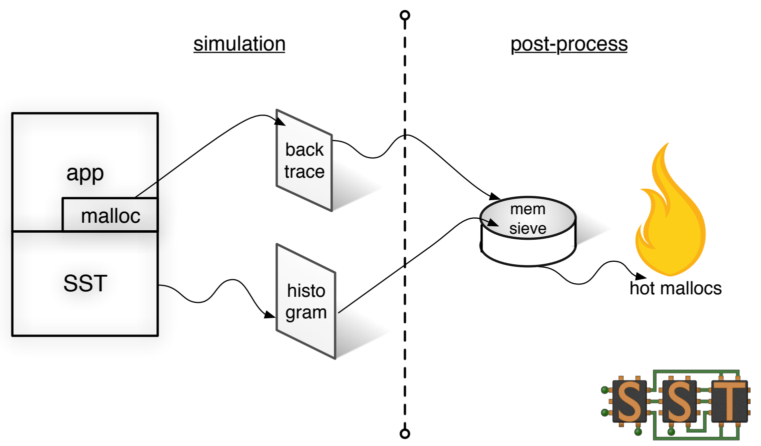


Figure 4.1: MemSieve tool

structure, MemSieve has some limitations. First, it does not model coherence and so MemSieve may be less accurate for applications that rely heavily on coherence, that is, have significant read-write sharing. Second, MemSieve implicitly assumes an inclusive last-level cache. For architectures where upper level caches (L1s for example) are exclusive or noninclusive of the last-level, MemSieve may underestimate the effective cache size. However, such architectures could be approximated by increasing the size of the MemSieve(s).

Validation

To validate that a MemSieve simulation yields similar memory patterns to detailed simulation, we compared the memory statistics from each simulation. This validation was done with the heavyweight architecture and compared to a MemSieve system with a single MemSieve of the same size as the aggregate of the heavyweight architecture's last level caches. Table 4.2 shows the read (left two columns) and write (right two columns) cache miss rates, measured as fraction of accesses that missed the entire cache hierarchy, for each application. The "Detailed" columns show rates for the detailed simulation while the "MemSieve" columns show the rates for the MemSieve simulation. For both reads and writes, the rates match closely, indicating that despite the reduced detail, MemSieve is able to accurately capture memory behavior.

Table 4.2: Validating MemSieve

Applications	Read Miss Rate		Write Miss Rate	
	Detailed	MemSieve	Detailed	MemSieve
PENNANT	1.15%	1.23%	4.81%	4.81%
HPCG	6.71%	6.71%	2.30%	2.31%
SNAP	0.42%	0.42%	0.71%	0.71%
MiniPIC	0.04%	0.04%	0.00%	0.00%

4.2.2 Analysis

Having validated that MemSieve accurately captures memory behavior, we now analyze each application using MemSieve. The first analysis looks at the overall characteristics of mallocs for each application. The second takes a look at how we can use MemSieve to determine how "ideal" an applications memory use pattern is, and the third looks at how the cache hierarchy of the light and heavyweight architectures affects that pattern. Finally, we do a small proof-of-concept, demonstrating that MemSieve can be used to realize substantial performance gains on real hardware.

Malloc Characteristics

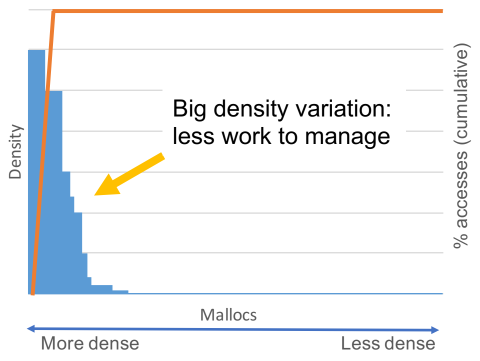
We begin by looking at some basic allocation characteristics for each application. These are shown for an 8GB data footprint in Table 4.3. The statistics cover execution from application begin through the end of the 'End' sample (see Table 2.2). SNAP is an exception because the MemSieve PIN tool encounters memory for the 8GB data set after the 'Middle' sample. Therefore, SNAP shows statistics from application begin through the end of 'Middle'. The first row, "Malloc count" shows the raw number of mallocs encountered during execution and the second row, "Malloc size" shows the total size of these mallocs. HPCG and MiniPIC have relatively fewer mallocs and the size of the mallocs matches the memory footprint indicating most data objects are allocated once and used throughout the application. In contrast, SNAP and PENNANT have both larger malloc counts and a much larger size than the memory footprint, implying frequent allocation and frees. This can cause problems, not only for manual MLM management where fewer allocations are easier to manage, but also in general. Frequent mallocs can reduce locality in the data set leading to worse cache and TLB performance, as well as increase overhead due to the individual malloc calls. We note that HPCG, prior to release 3.0 which switched to allocating arrays in full rather than element-wise, exhibited the same behavior and suffered lower performance (both in simulation and on hardware).

Table 4.3: Malloc characteristics

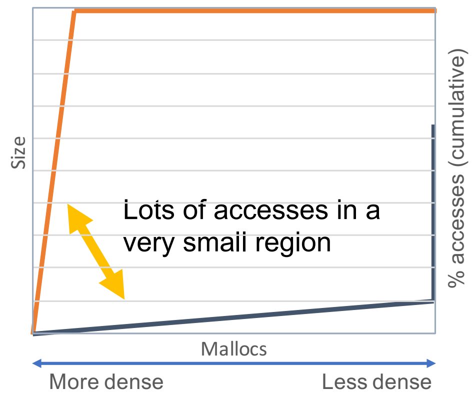
	PENNANT	HPCG	SNAP	MiniPIC
Malloc count	8B	23M	1B	438K
Malloc size	32.1 TB	7.43 GB	30GB	7.9GB
Distinct traces	248	612	323	39043
Accessed traces	140	146	90	10794
Size of accessed traces as % total	89.7%	99.987%	89.6%	84%
App traces	220	583	188	38999
Accessed app traces	129	132	58	10781

The next two rows, "Distinct trace" and "Accessed traces" show the number of distinct call traces among the mallocs and the number of distinct call traces that are accessed in main memory (as opposed to cache or not at all). We look at the entire call path rather than just the malloc call site to enable us to differentiate mallocs that are part of separate data structures that share the same code. For example, in HPCG, we can differentiate mallocs for each level of the multigrid, although the code building each level is shared. Looking at distinct traces, we see that PENNANT, HPCG, and SNAP have low counts. For manual MLM management, this means there are relatively few paths that a programmer would need to consider, simplifying the code changes. For MiniPIC, there are many distinct call paths, complicating the programmer's job. For such an application, manual allocation would almost certainly require assistance from an automated tool, like MemSieve, to identify mallocs of interest. Even so, the programmer may need to mark many distinct traces for allocation in HMC, requiring more pervasive code changes than for the other applications.

Turning to the "Accessed trace" row, the most common reason a trace does not have memory accesses (cache misses) is that the malloc called by that trace is small and frequent allocations and frees to the memory returns the same memory address. The address ends up cache resident and does not incur cache misses. The next most common cause is that the mallocs occur in startup, before we begin profiling accesses in the simulation. Least often, the mallocs occur during non-sampled iterations of the main computation. The next row in the table, "Size of accessed traces", shows that while in many cases less than half of the distinct traces are accessed, the size of the mallocs associated with the accessed traces accounts for 84% to almost 100% of the total allocated size shown in row 2. Thus, the sampling methodology described in Section 2.1.1 is sufficient for profiling a high majority of mallocs.



(a) Density and access graph for an ideal application



(b) Size and access graph for an ideal application

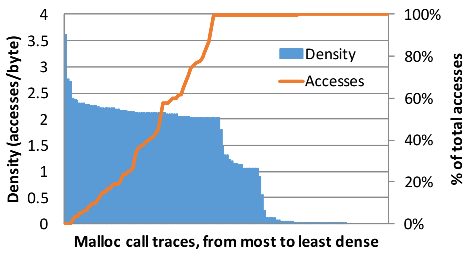
Finally, the last two rows in the table show the count of distinct traces ("App traces") and the number of those with accesses ("Accessed app traces") if we remove library calls from the call traces. Removing library calls has two effects. First, it removes mallocs reached entirely through library calls, such as occur before program start. These mallocs are not accessible to programmers anyways. Second, removing library calls enables us to merge traces that have the same call trace through the application but differ once they enter libraries. Because an application's separate paths through libraries are generally opaque to programmers, this gives the "real" number of call traces a programmer will need to contend with.

Ideal Malloc Behavior

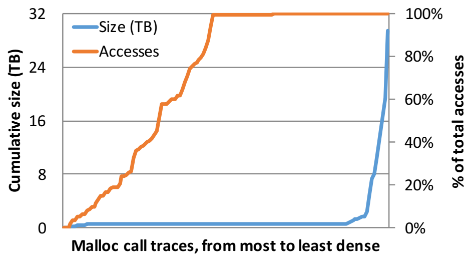
In this section we take a deeper look at the malloc behavior of each application and compare it to an "ideal" application whose memory would be easy to manage manually. Because we expect that application writers will be managing mallocs according to their unique call traces, we consider the densities and sizes of mallocs at that granularity. We begin by describing this ideal application. The first measurement we look at is density. Recall that density is the number of memory accesses to a region divided by its size. An ideal application has very few, very dense regions that account for most of the application's memory accesses. In this case, the programmer would have few malloc call traces to manage, the candidate call traces would be easily distinguishable from the rest due to their high density, and placing mallocs generated by those call traces into HMC would mean that most of the application's accesses benefit from the higher bandwidth memory. The density and access graph for such an application is shown in Figure 4.2a. Density is plotted in the blue bars on the left Y-axis for each unique malloc call trace sorted from most to least dense along the Y-axis. The cumulative fraction of accesses accounted for by each call trace is shown by the orange line and plotted on the right Y-axis.

The second measurement we consider is size of the mallocs associated with each call trace, as shown in Figure 4.2b. The blue curve shows the cumulative size (left Y-axis) for each call trace sorted from most to least dense along the X-axis. Cumulative accesses are again shown by the orange curve and plotted on the right Y-axis. Because the size curve grows slowly while the access curve grows quickly, this application will need less HMC capacity to capture the majority of memory accesses. However "less capacity" is a function of the total allocated size and so one must consider both the scale of the left Y-axis as well as the growth of the size curve.

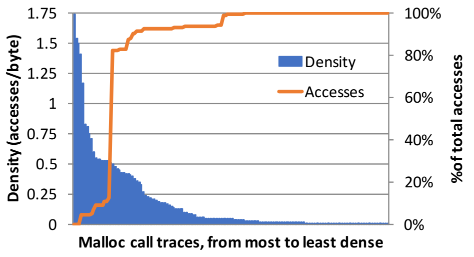
Having described an ideal application, we turn to the focus mini-applications. PENNANT's density and size graphs are shown in Figures 4.3a and 4.3b respectively. Looking at the density graph, we can see the PENNANT differs greatly from the ideal application described above. Nearly half of the mallocs are equally dense and accesses are spread evenly across those mallocs, meaning the programmer is going to need to manage many allocations to take advantage of the HMC's higher bandwidth. Because so many mallocs are



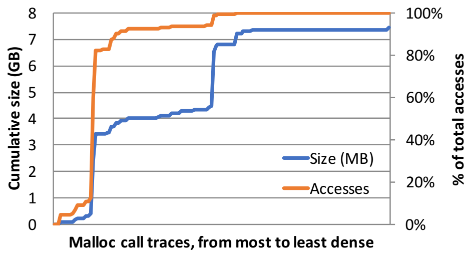
(a) Density and access graph for PENNANT



(b) Size and access graph for PENNANT



(a) Density and access graph for HPCG



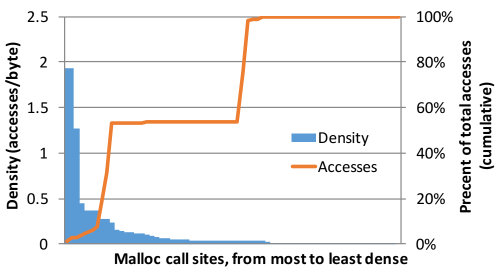
(b) Size and access graph for HPCG

dense, it is likely that dynamic migration between the HMC and DRAM will be be necessary. However, without looking at the behavior over time we cannot tell with certainty how the allocations are accessed in different application phases. MemSieve does have the ability to profile over time, but for this initial study we did not pursue it. The second, size graph, is better at first glance as the many dense mallocs account for a small fraction of the overall size. Still, because the total size of mallocs is 30TB, a small fraction may not be small enough to fit in HMC.

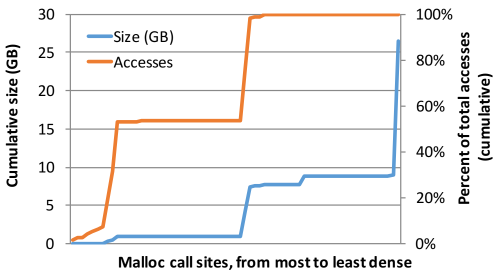
In contrast to PENNANT, HPCG demonstrates nearly ideal behavior. Its density graph, shown in Figure 4.4a has a steep cliff, far to the left indicating that the programmer will need to manage very few allocations to capture both the majority of dense mallocs and the majority of accesses (shown by the red line). The size graph (Figure 4.4b) is less ideal. Although the most dense mallocs account for just 1/8th of the total memory, the programmer may not be able to fit all of the moderately dense mallocs in HMC. This means slightly lower performance especially at smaller HMC capacities, but compared to PENNANT, HPCG will require less work (and potentially no dynamic migration) to achieve the performance.

Figure 4.5a and Figure 4.5b show the density and size graphs, respectively, for SNAP. The density appears ideal however, the access curve resembles that of PENNANT – the densest mallocs account for a moderate fraction of overall accesses. Looking to the size graph we see that these densest mallocs are also quite small, about 1GB in total. As such it is likely that all of the dense mallocs will fit in even a small HMC. However, as the dense mallocs capture only about 50% of the accesses, steering the rest of the accesses to the HMC will require considering sparser mallocs. Further, the size graph indicates that while most mallocs are small (i.e., flat curve), there are a few large, sparse mallocs and these correspond to large jumps in the access curve. Placing large, sparse mallocs in a limited size HMC will be difficult and therefore for SNAP, one may need to consider a smaller granularity than mallocs to capture more accesses. Overall SNAP presents a mixed picture. One should be able to easily fit the densest mallocs into HMC, but doing so will only capture half of the accesses. Capturing the remaining accesses will require more work.

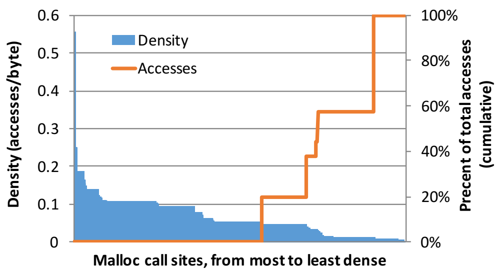
Finally, we analyze MiniPIC's allocation behavior. Figure 4.6a shows MiniPIC's density graph. Of the applications analyzed, MiniPIC is the least ideal. The majority of dense mallocs account for very few accesses and so placing the dense mallocs in HMC is likely to have no effect on performance. The size graph (Figure 4.6b is similarly non-ideal. The size and access curves are both far to the right, indicating that not only must one consider the sparse mallocs to capture the majority of memory accesses, the sparse mallocs



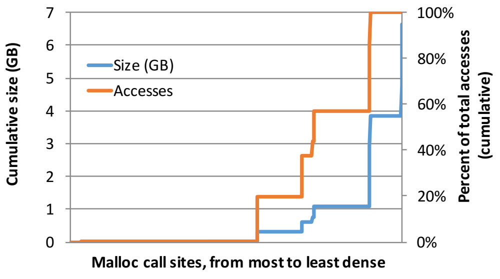
(a) Density and access graph for SNAP



(b) Size and access graph for SNAP



(a) Density and access graph for MiniPIC



(b) Size and access graph for MiniPIC

are large and therefore likely difficult to place in HMC. While MiniPIC showed no performance improvement in Section 3.1 and therefore we do not consider it further, these graphs indicate that even if it did benefit from HMC, it would not be easy to capture the performance with a small HMC-to-DRAM ratio.

Architecture Effect on Dense Mallocs

Our last analysis looks at the effect of the cache architecture and application input on which mallocs are considered dense. For manual allocation this is important, as the programmer would need to redo the allocation management if it varied with either architecture or input. Input is arguably a bigger problem, as applications are likely to be run on few architectures, but the inputs may vary widely. Still, variance due to either architecture or input increases the work of the programmer and makes manual allocation less appealing.

Figures 4.7 and 4.8 show the variance in the top malloc trace ranking for PENNANT and HPCG between two data set sizes. Similarity on the Y-axis is measured as the fraction of the top N malloc traces (for N shown along the X-axis) that are the same for both the smaller and larger data footprint. For example, a 50% similarity at N=2 would mean that one of the top two traces match in the rankings for the smaller and larger data footprint. While both graphs show some differences in the top 20-30, overall the sets of dense traces are similar. For perspective, in the next section where we manually place mallocs, we generally consider the top 40-50 traces. Thus, while for very small HMCs the variation may have effect, for larger HMCs like those studied here, the differences in ranking do not greatly affect malloc placement. This is advantageous as it means a programmer can profile and the application once and apply the results to a wide range of inputs. For HPCG at N larger than 60, the similarity does start to drop off. This occurs because the malloc traces ranked ¿60 tend to have few accesses (e.g., less than ten) and small simulation variations can be the difference between the trace being cached or not.

We now look at the variance due to architecture. For this experiment we ran HPCG and PENNANT using MemSieve representations of both the lightweight and heavyweight cache hierarchy. Figure 4.9 shows the resulting variance for PENNANT and Figure 4.10 shows HPCG's variance. As in the previous set of graphs, the X-axis shows the number of malloc traces compared and the Y-axis shows the percent of malloc traces in each of the lightweight and heavyweight ranking that match. For PENNANT and HPCG there

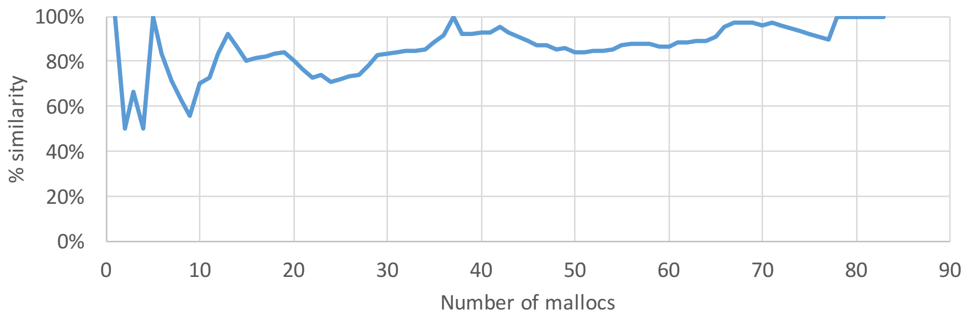


Figure 4.7: PENNANT: similarity in malloc trace ranking between a 1GB and 8GB data footprint

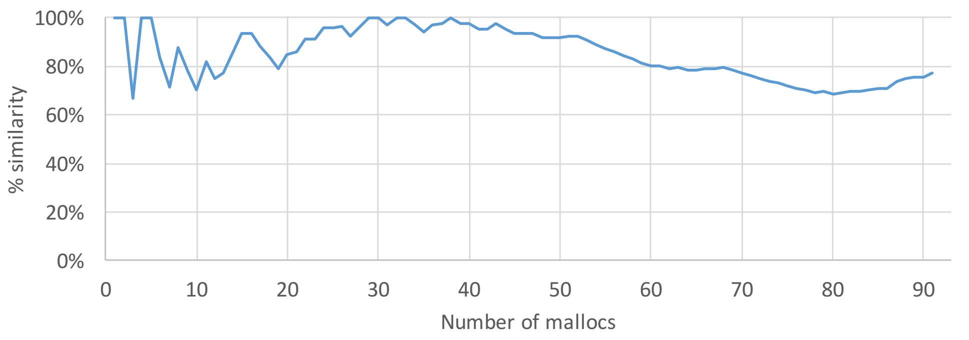


Figure 4.8: HPCG: similarity in malloc trace ranking between a 4GB and 8GB data footprint

is some difference in the top 10 to 20 traces, but as the number of traces considered grows, the ranking is similar. In both, similarity reaches 80% or greater by N=20. This is well within the 40-50 traces considered for manual allocation and therefore the initial differences are unlikely to affect overall placement.

Dense Mallocs: What are they?

The previous sections gave an overview of the density distributions and characteristics of each application's mallocs. In this section, we look deeper at those mallocs and identify precisely what they are in reference to the application code. For PENNANT, the dense mallocs tend to be the arrays found in the Hydro class, although a few of the Mesh arrays are dense as well. For HPCG, the densest mallocs are the CG vectors,

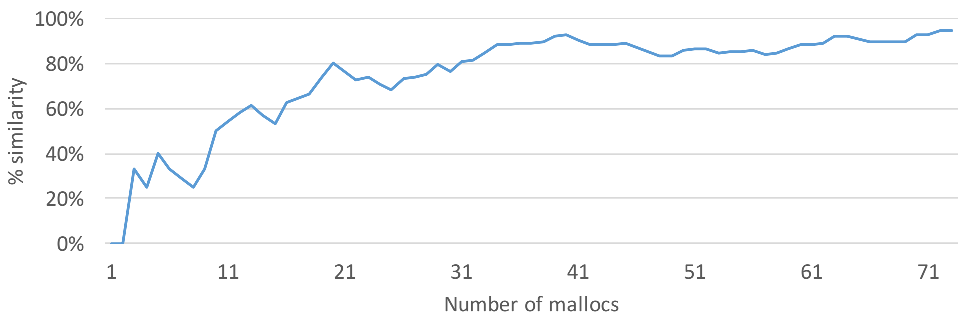


Figure 4.9: PENNANT: similarity in malloc ranking between the heavy and lightweight architectures

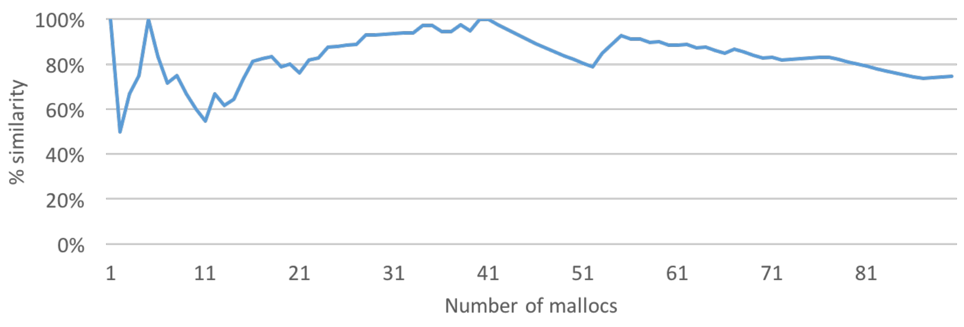


Figure 4.10: HPCG: similarity in malloc ranking between the heavy and lightweight architectures

followed by some of the structures associated with levels of the multigrid. In general, finer levels of the grid have denser mallocs than coarser ones. But, this is not a strict rule. For example the textitAxf vector is one densest mallocs regardless of grid level.

In SNAP, many of the densest allocations are working arrays allocated in the *Solvar* module. Other dense allocations include the leakage arrays and flux moment arrays. Finally, in MiniPIC the *ParticleTypeList* is the densest allocation, followed by some of the face and edge maps. These include the *boundary_face_map*, *owned_boundary_edge_map*, and *owned_face_map*.

Translating MemSieve to Hardware

We envision that the MemSieve tool will be useful for programmers trying to determine which allocations are the best candidates for HMC (or HBM) allocation. In this section we take a look at that workflow and validate that MemSieve's suggestions do result in performance improvements on real hardware. For this study we looked at MiniFE rather than one of the four focus mini-apps. The major reason for this is that MiniFE is simpler than the focus mini-apps so that manually editing it for allocation on real hardware is easier. Running MiniFE through MemSieve revealed that four of the top five ranked mallocs were Vectors. The sixth-ranked malloc was the array, which is likely too big to place in the Knight's Landing's HBM. The remaining top ten mallocs were comprised of calls related to OpenMP for loops which are difficult to manually allocate since the allocation is done by the OpenMP library. To test MemSieve's suggestions therefore, we manually edited MinIFE to place the Vectors in HBM. Figure 4.11 shows the results. On the left is the performance of MiniFE on the Knight's Landing when running completely out of the DRAM. Performance is normalized to this configuration. In the middle is the performance of MiniFE when running completely out of the HBM. While this is the ideal case, the HBM is too small for realistic problem sizes to fall in this category. Therefore, a hybrid approach is needed, as shown on the right. This bar shows the performance of MiniFE when the Vectors, per MemSieve's recommendation, are manually allocated to HBM. In this case, we see a significant, 1.5X speedup over DRAM only.

Processing-In-Memory and MemSieve

We finish our discussion of MemSieve by touching briefly on the subject of processing-in-memory (PIM). Because HMCs have a logic layer under the DRAM stack, they are ideal candidate technologies for implementing PIM. However, the question is not the capability of HMCs to support PIM, but rather what kind of PIM applications might benefit from. One way to approach this question is to look at which application allocations are likely to be HMC-resident. Allocations that are largely cache-resident or move between caches and memory would increase PIM overhead by requiring extra flushes and coherence actions to ensure coherent PIM. On the other hand, allocations that are likely to be resident in DRAM are not candidates for PIM as operating on them would require migrating the data to HMC. Fortunately, MemSieve can answer this

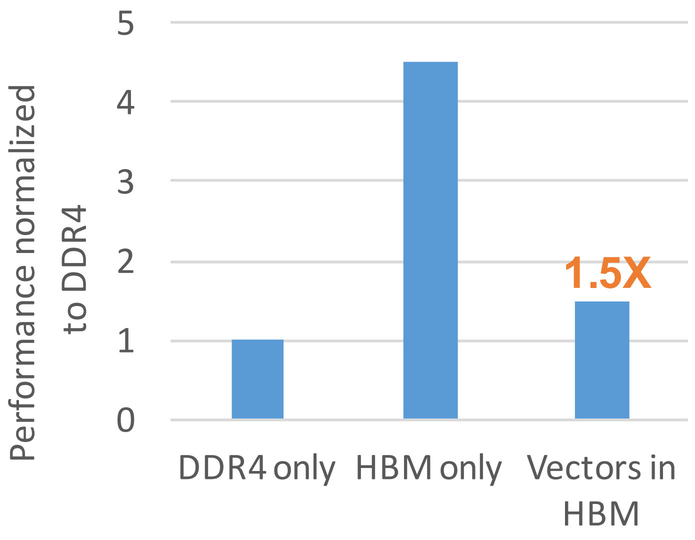


Figure 4.11: Performance of MiniFE on the Knight's Landing architecture

question with its dense allocation ranking. Importantly, we note that MemSieve's ranking is not limited in usefulness to manual allocation. An intelligent OS/runtime approach and the hardware caching policies that we'll discuss in the next chapter are all methods for identifying dense allocations (or pages). While they may not be as predictable or efficient as manual allocation, well-designed automatic management schemes should on average end up with the same allocation as the manual approach. Thus regardless of the management scheme used for MLM, we can use MemSieve to answer the question of what is likely to end up in HMC and therefore where the application might benefit from PIM without incurring extra overhead.

As an example, one common allocation type that shows up across all the applications is OpenMP loop pragmas. Unsurprisingly, these figure especially heavily in the lightweight architecture's malloc ranking as that architecture has many more threads. Because these mallocs tend to be small they are likely to fit in HMC even though many are only moderately dense. As such, one suggestion for PIM would be to add PIM computation capability for some of the small OpenMP loops with no child calls.

4.3 Comparing Software Allocation Strategies

In the previous section we discussed the role of MemSieve in steering manual allocation as well as for analyzing the likelihood that an application will benefit from a given management policy. In this section, we compare the four policies outlined at the beginning of the chapter and discuss why they perform as they do. To recap, those four policies are (1) greedily allocating pages in HMC based on first touch, (2) greedily allocating mallocs in HMC as they are encountered, (3) using MemSieve to analyze the application and statically allocate the most dense allocations to HMC, and (4) using MemSieve to analyze application phases and dynamically migrate allocations so that the most dense allocations for each phase are in HMC. Because MiniPIC did not show any potential for improvement (Section 3.1), we do not analyze it here and instead focus on PENNANT, HPCG, and SNAP. Further, as the heavy and lightweight architecture perform similarly, we look primarily at the heavyweight architecture. However, we will compare the two architectures in Section 4.3.3.

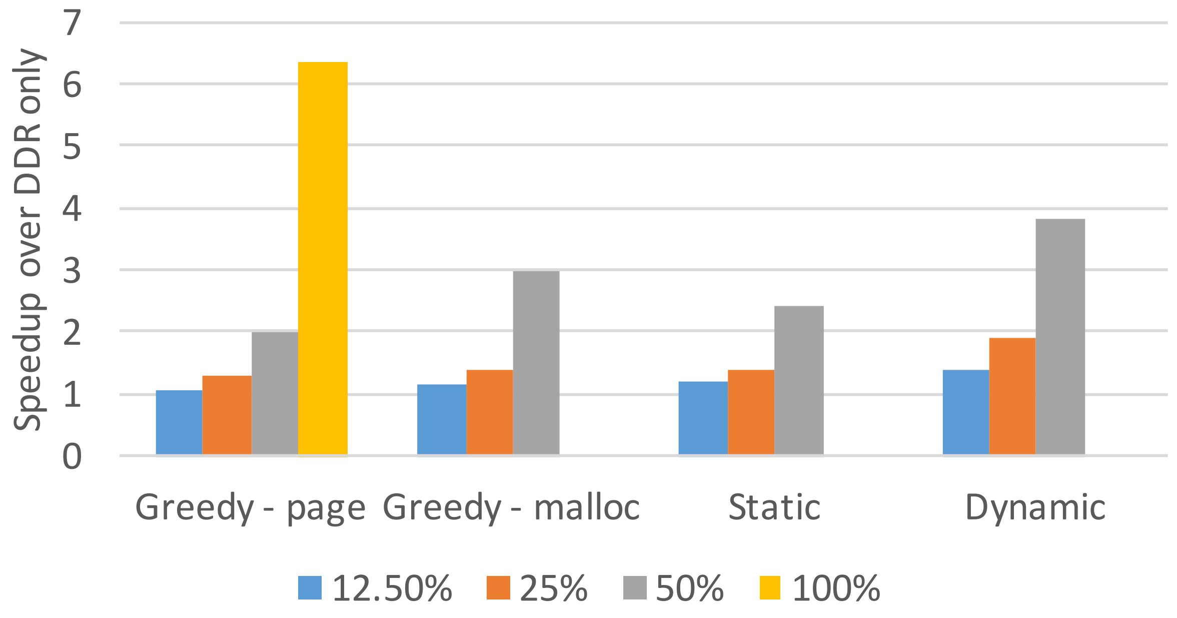


Figure 4.12: Software allocation policy performance for PENNANT

4.3.1 Comparing Software Approaches

Figure 4.12 shows the performance of the software allocation strategies on the X-axis for PENNANT. *Greedy-page* and *Greedy-malloc* are the static greedy page-based and malloc-based approaches respectively. *Static* is the static manual approach and *Dynamic* is the dynamic manual approach. We normalize performance to the performance of DRAM-only (Y-axis) and show results for three HMC capacities ranging from 1/8th to 1/2 of PENNANT's data footprint. In addition, for reference we include the case where all memory is HMC (yellow bar). Since allocation strategy is irrelevant in the all-HMC case we show it only for Greedy-page. Looking at the graph, we first conclude that PENNANT performs much better with a larger fraction of HMC. This fits with the MemSieve analysis which showed that PENNANT's dense mallocs had a large memory footprint. Second, we observe that the first three strategies, Greedy-page, Greedy-malloc, and Static, perform similarly, while Dynamic roughly doubles the performance improvement. Again, the MemSieve analysis showing many equally-dense allocations predicted this outcome. Unfortunately, this means that for best performance, PENNANT will require more HMC and dynamic migration, the most complex management policy.

Next we turn to HPCG, shown in Figure 4.13. Again, the X-axis shows the allocation strategies and the Y-axis shows performance normalized to DRAM-only. For reference, because MiniFE and HPCG do similar computation, the star to the right of the Static bars shows the performance on Knight's Landing hardware of MiniFE with Vectors statically allocated in HBM (Section 4.2.2). As with PENNANT, there is a large jump in performance, especially from 25% to 50% HMC. Looking back at the MemSieve analysis in Section 4.2.2, one sees that while the densest allocations comprise a small fraction of the data footprint, there is a steep rise in the percentage of accesses accounted for that coincides with a rise in the malloc size curve from about 12.5% of the footprint to nearly 50% of the footprint. With the exception of Greedy-malloc, the policies perform similarly. We believe that Greedy-page happens to get a good mapping because the most dense allocations tend to be accessed early in an iteration (i.e., the CG vectors and lower levels of the multigrid).

Lastly, Figure 4.14 shows the results for SNAP. This graph differs from the previous two applications in that the fraction of HMC matters little to performance, and *Greedy-page* rather than one of the malloc-granularity policies performs best. This occurs because SNAP has two very large mallocs that each account for 42% of the data set size. The mallocs are only moderately dense, but once the HMC capacity is small

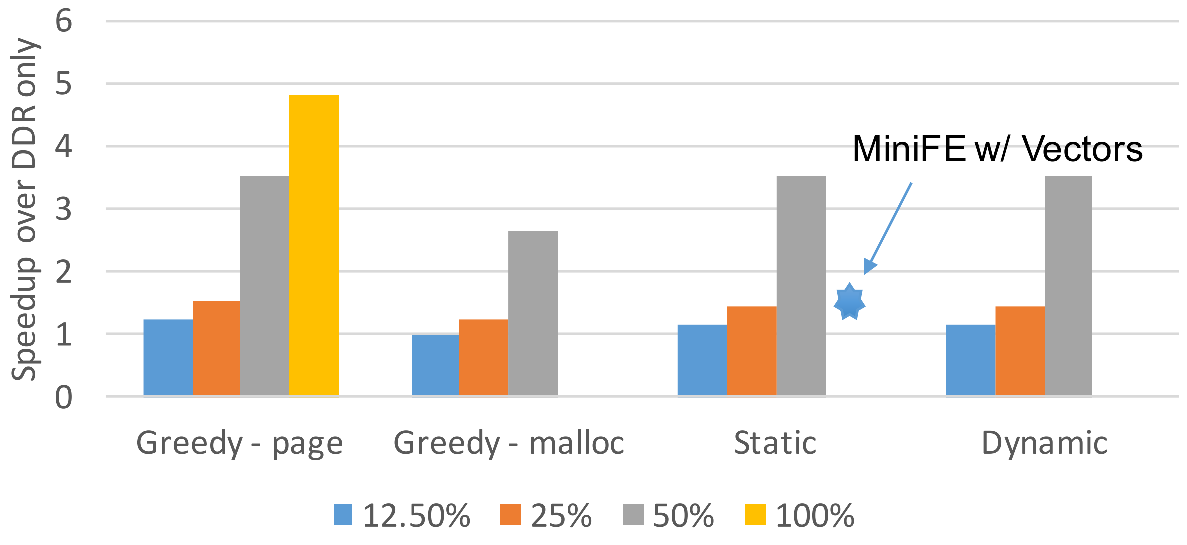


Figure 4.13: Software allocation policy performance for HPCG

enough that neither fits, the malloc policies capture the same mallocs. The page-based policy on the other hand is able to capture part of these mallocs, resulting in better HMC utilization and higher performance. This points to a need for a code change if the malloc-policies are to perform well. Large mallocs lead to inefficient use of the HMC. It is also possible that large mallocs hide smaller dense and less dense regions within them.

Overall, the greedy policies perform well, but this statement comes with an important caveat. The studied mini-apps, while more complex than many, are not real applications. Full applications with many execution phases and allocations that occur throughout execution are unlikely to perform well with greedy because only the early allocations and phases will benefit from the HMC. PENNANT hints at this case, where allocations occur and the set of dense data structures changes throughout execution. As such, support for dynamic migration is essential, whether application- or OS-managed.

4.3.2 Effectiveness of Manual Allocation

We now look slightly deeper at the effect of manual allocation on the spread of accesses to the different memory levels. Table 4.4 shows the fraction of memory accesses that went to HMC instead of DRAM for each of the benchmarks, policies, and HMC capacities in the preceding section

Table 4.4: Fraction of accesses to HMC compared to DDR

Policy	Greedy-page			Greedy-malloc			Static			Dynamic		
Capacity	12.5%	25%	50%	12.5%	25%	50%	12.5%	25%	50%	12.5%	25%	50%
SNAP	34%	47%	68%	38%	52%	52%	36%	52%	49%	40%	44%	44%
HPCG	26%	49%	92%	5%	29%	75%	23%	47%	93%	23%	49%	93%
PENNANT	20%	34%	62%	15%	30%	55%	20%	33%	67%	38%	47%	87%

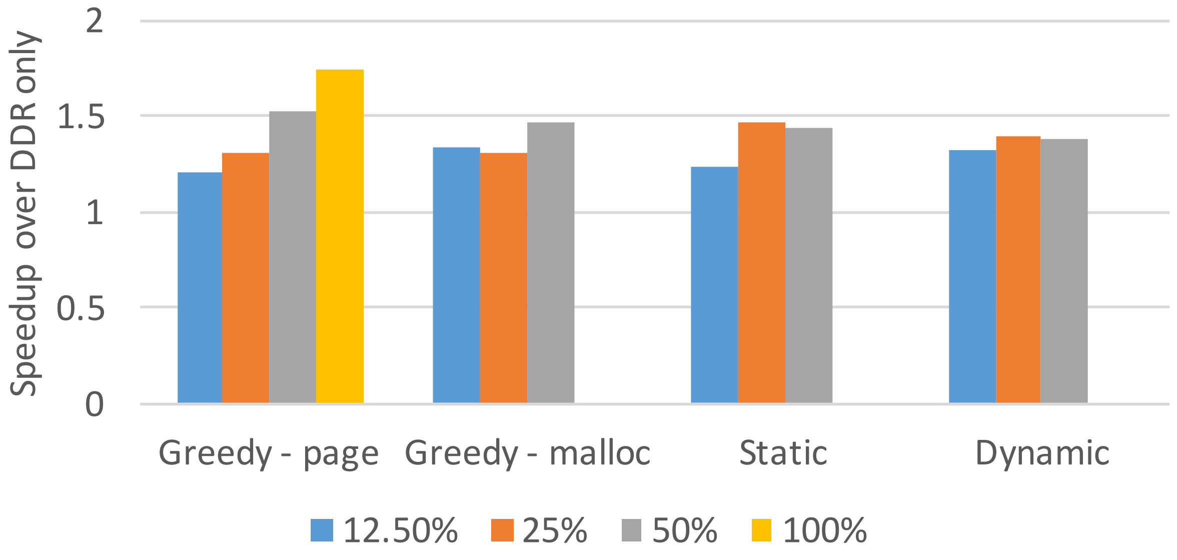


Figure 4.14: Software allocation policy performance for SNAP

4.3.3 Effect of the Architecture

The previous sections showed results for the heavyweight architecture only, we now look at the lightweight as well. Because the potential improvements from HMC were similar for both architectures (Section 3.1), we look only at PENNANT. Figure 4.15 shows PENNANT's performance (Y-axis) for the different capacities (bars) and policies (X-axis). Comparing these to Figure 4.12 above, we see the same trend. The first three policies perform similarly, while dynamic outperforms the others by nearly a factor of 2X. Again, there is a large jump in performance between the 25% and 50% capacities.

4.3.4 Conclusions

Together, the MemSieve analysis with the analysis of software managed MLM strategies define a path forward for software managed MLM. Because the simple, OS-managed greedy strategies performed fairly well, we argue that the best path forward for MLM management is likely to be an OS-managed approach that uses programmer hints for intelligent allocation. We think that for more complex applications with many execution phases, the OS will not "get lucky" with a static greedy allocation and so a hybrid approach will be necessary. This approach should include support for dynamic migration. While the wholly manual approaches performed well, they come with numerous drawbacks including lack of support for legacy codes, lack of programmer control over library allocations, and no centralized management to handle the complexities of sharing HMC capacity between library and application code.

A second conclusion we draw is that the best performance will likely require algorithmic changes. In this study we see that SNAP cannot easily take advantage of the HMC at a malloc granularity because its mallocs are too large. Breaking up the malloc would likely improve performance. Designing the application's execution phases so that a single phase can operate mostly out of HMC would also be beneficial. For PENNANT, the large phases still spill out of HMC and for HPCG, the phase granularity was too great.

Finally, we observed similar application memory behavior across the very different heavy and lightweight architectures. This result is not unexpected given that the underlying memory technologies are the same, the allocation malloc behavior doesn't vary greatly with architecture, and the potential HMC performance was also similar. However, it indicates that these results are likely applicable to a variety of architectures

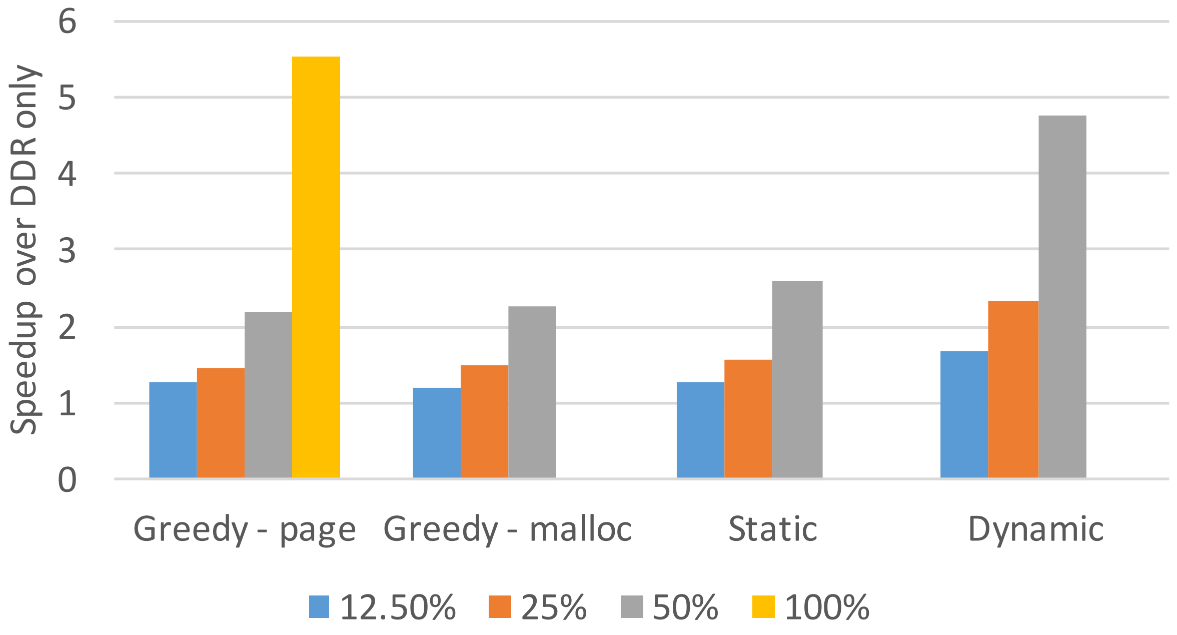


Figure 4.15: Software allocation policy performance for PENNANT on the lightweight architecture

beyond those studied.

Chapter 5

Automatic Management

Manual management is advantageous because it requires no modification to the hardware, however it does require programmer effort and / or knowledge. Thus, it is worth exploring MLM which is managed without any programmer effort. In automatic management, the hardware, operating system, runtime, or some combination thereof move data between the levels of memory without user guidance.

In this chapter, we explore several automatic MLM management policies.

5.1 Background

5.1.1 Related Work

There is a wealth of related work on automatic paging algorithms and cache policies. For caches, most works emphasize different replacement policies, such as Least Recently Used (LRU), or pseudo-LRU for SRAM caches[7]. A variety of virtual memory paging schemes have been explored such as simple FIFOs, LRU, multiple queue systems[9], or adaptive models[3].

Multi-level Memory has explored the combination of DRAM and NVRAM[17] with demand paging, sometimes using dynamic policies[12]. [16] explores hybrid policies to reduce energy. In particular, it explores using historical page access information (HRank) to performs bulk swaps at regular intervals and an inclusive demand paging policy (PCache). Other work[1] examines user-directed management of MLM.

5.1.2 Key Differences

This work differs primarily in that it focuses on the **addition** policy rather than the replacement policy. In most systems (e.g. SRAM caches backed by DRAM or DRAM backed by hard disk) the relative latency of the faster storage is so much lower than the backing store that the best addition policy is almost always a simple “always add” policy. In an MLM system comprised of DDR + fast stacked memory that is not the case.

The MLM management problem is a subset of the general caching / paging problem. This problem consists of two parts: addition policies, that determine when a block of data is moved to the faster memory, and replacement policies, that determine when a block of data in the faster memory should be moved to the slower memory. The MLM problem is similar to the management of processor SRAM caches or virtual memory (VM). However, due to the massive differences in latency between levels, both SRAM caches and VM must bring data into the closer memory before it is used. In these systems, the addition policy could be described as “Always Add.”

In MLM, the latency of the Fast memory may not be dramatically different from traditional DDR DRAM. Thus, there are many cases where it may make sense to leave a data block in the slower memory if it is not going to be used frequently. This saves the overhead of transferring between levels of memory. Early simulations of several benchmarks of a system with 128 pages of fast memory using a simple always add policy found that performance was reduced by 9% to 61% (average: 52%) poor policies can turn MLM from a benefit into a liability.

5.2 Early Analysis

To develop potential policies, we began by analyzing several applications' (Section 2.1) memory access patterns.

5.2.1 Access Patterns Within A Page

Using the infrastructure created for [8], we examined how post-cache memory accesses are distributed within a page. The analysis filter records each access to a page and compares it with the last access to that page. It then computes the offset ($\Delta = Addr_{new} - Addr_{old}$) and classifies the new access into one of five categories: Same Address ($\Delta = 0$), next cache line, $\Delta = 1$, 'jump forward" ($\Delta > 1$), previous cache line ($\Delta = -1$), or "jump back" ($\Delta < -1$).

Each page was placed in a five dimensional space based on the percentage of memory accesses that fall into each category. A *k*-means clustering algorithm was applied to the classified pages for each application to identify "typical" pages. Figure 5.1 shows the center points of four clusters for the access patterns for three applications. For each center, the graph shows the five "directions" that the next memory access takes ($\Delta =$ 0,1,-1,>1,<-1) and the percent of memory accesses that fall into that cluster.

For example, in Lulesh 64.1% of accesses go to pages which are dominated by an access pattern that is overwhelmingly incrementing through memory, but there are a seizable number of accesses (26.9%) which go to pages that *tend* towards increment, but have a third of their accesses either jumping ahead or behind by some amount. In contrast, MiniFE is almost completely dominated by pages that are simply incrementing through memory.

This indicates that many applications have a significant "streaming" component – large number of pages which are accessed sequentially. Because so many pages have a strong $\Delta = 1$ component, it should be possible to detect "streaming" pages by noting a relatively short series of consecutive positive incremental accesses.

5.2.2 Multithreaded Access

We also wished to examine how many threads of execution access a given page. The goal of this analysis was to determine suitable page sizes. Ideally, a page size should be large enough that per-page meta-data is limited, but small enough that patterns (e.g. scans) are preserved and swap time is limited. We theorize that minimizing the number of threads that are accessing a page would be beneficial. If fewer threads are accessing a page, post-cache hardware is more likely to identify simple patterns (e.g. linear scans). Additionally, a page with fewer threads accessing it can probably be swapped from one level of memory to another without disrupting as many threads.

Using SST, we performed an analysis for several mini-apps. For each page in memory we looked at how many unique threads accessed that page, and how many times each thread accessed that page. For this analysis we only examined post-cache accesses (i.e. main memory accesses).

Across CoMD (Figure 5.2(a)), Lulesh (Figure 5.2(b)), and MiniFE (Figure 5.2(c)) 4KB pages are accessed by only a small number of threads – generally one or two threads. Even in the case of Lulesh, which has a

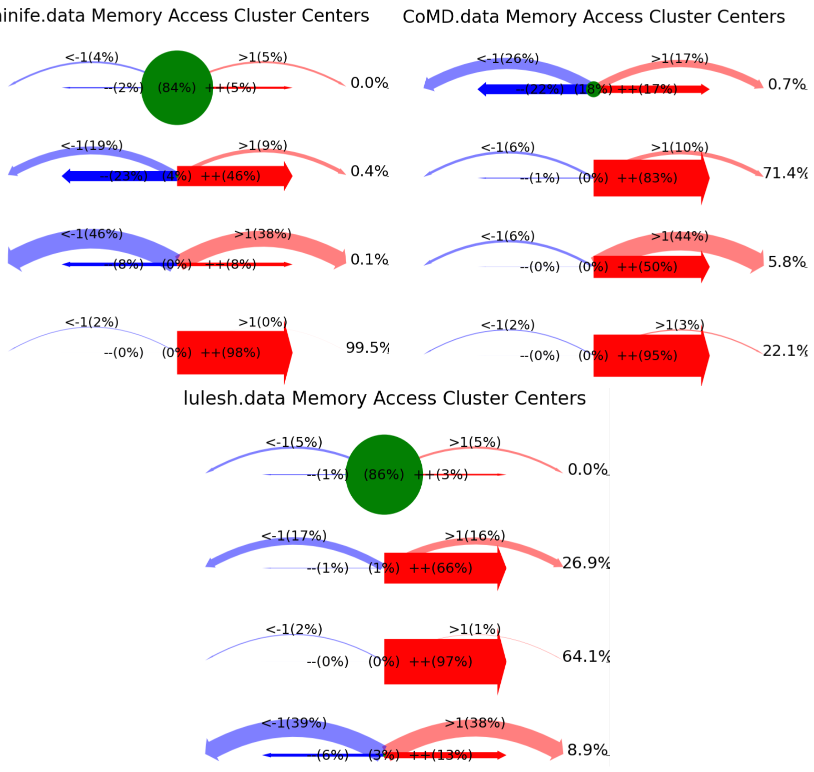


Figure 5.1: Intra-page access patterns

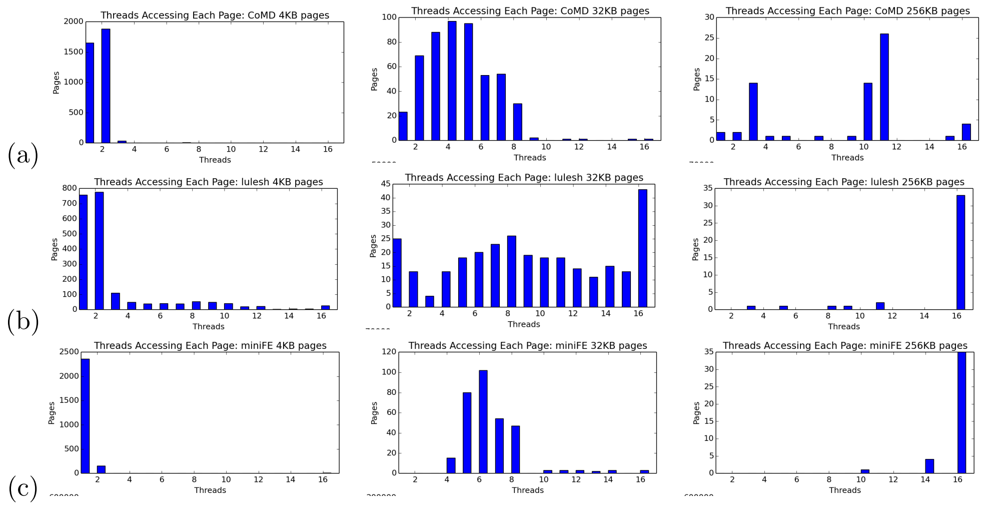


Figure 5.2: Thread use per page for (a) CoMD (b) Lulesh and (c) MiniFE

sizable number of pages accessed by multiple threads, the vast bulk of accesses go to pages that are touched by only one or two threads.

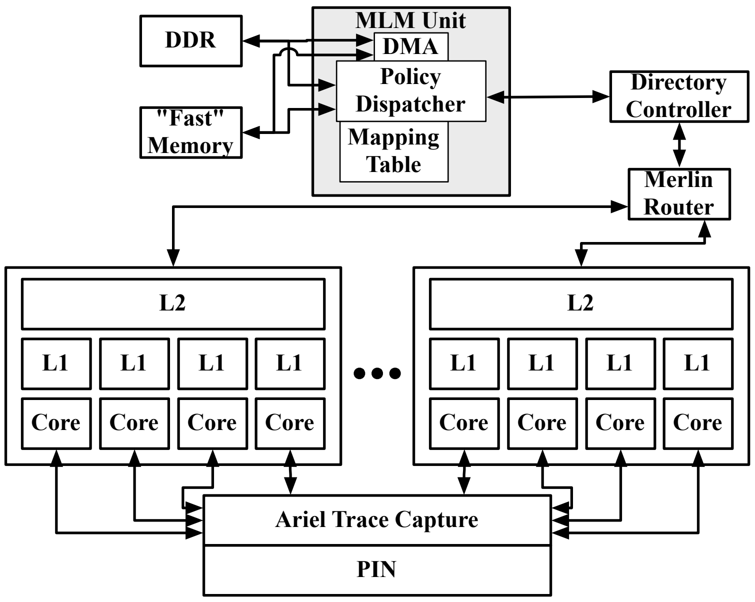


Figure 5.3: Proposed architecture and simulation model

5.3 Policies

To explore multiple MLM policies, we propose a simple, self-contained architectural block, the MLM Unit, which dispatches main memory accesses to either traditional DDR DRAM or a “Fast” memory. Figure 5.3 shows the proposed block, which is comprised of three subunits:

- **Policy Dispatcher**: The control unit of the block, which decides if a given memory page should be located in the DDR or fast memory, swaps pages as needed, and dispatches incoming memory accesses appropriately. The addition and replacement policies are detailed below (Sections 5.3.1 and 5.3.2).
- **Mapping Table**: A TLB-like structure (in the sense that it contains metadata about each memory page) which records which pages reside in fast memory, as well as policy-specific meta-data for each page.
- **DMA**: A unit to transfer pages between the DDR and fast memory without involving the CPU or coherent cache hierarchy.

The MLM Unit attaches to the memory hierarchy after the directory controller, so all coherency decisions have already been made. Because there is only one valid copy in the backing store (either in the DRAM memory or “Fast” memory), the MLM Unit does not require additional coherency tracking, nor does the cache coherency protocol need to be modified to support MLM. Additionally, the mapping table data is only used locally and does not require complex management like a TLB.

Though the DMA subunit and certain updates to page metadata require hardware support, the actual policy decisions could be made purely in hardware or with more explicit software, operating system, or runtime control.

5.3.1 Addition Policies

Addition policies determine if a candidate page which is being accessed should be added to the “fast” memory. Seven addition policies were implemented. Many of these policies require a count of accesses be kept for each page in memory. This count, *touchCount*, is reset every 5ms. The policies we examined are:

- **addT** (Threshold) is a simple threshold policy. If the candidate page's *touchCount* is greater than a given threshold it will be added to the "fast" memory.
- **addMFU** (Most Frequently Used) If a candidate page's *touchCount* is greater than the lowest *touchCount* of all pages currently in fast memory, the candidate page will be added. The overhead of this policy would probably be prohibitive in practice, but it is included for completeness. This policy is similar to HRank [16], but is performed continuously and for individual pages instead of multi-page transfers at timed intervals.
- **addRAND** (Random) Each time it is touched, the candidate page has a 1-in-8192 chance of being added. This probability was chosen after a search of parameters to give good performance for the given applications.
- **addMRPU** (More Recent Previous Use) If the candidate page's last touch was more recent than the least previously used page in fast memory it is added. This requires that each page in memory be marked with the time of its last access.
- **addMFPRU** (More Frequent, More Recent Previous Use) As addMRPU, but only add the candidate page if it also has a higher *touchCount* than the least recently used page in fast memory.
- **addSC** (Stream Conscious) A policy inspired by the access patterns detected in Section 5.2.1, addSC is similar to addT, but also detects streaming pages. If the last six accesses to a page have been to sequential cache line addresses (i.e. $\Delta = 1$), then it only has a 1-in-1024 chance of being added. This would require additional metadata (sequential access count, last cache line address) to be stored for each page.
- **addSCF** (Stream Conscious-Frequency) Operates as addSC, but only add the candidate page if it also has a higher *touchCount* than the least recently used page in fast memory.

In addition, all policies include "swap throttling" in which they will not start swapping a page to fast memory if the DRAM transaction queue is full.

5.3.2 Replacement Policies

Replacement policies determine which page should be removed from "fast" memory to make room for a new page. We looked at six policies:

- **FIFO** (First In / First Out) When a new page is added to fast memory it is added to the front of a list. When a page needs to be removed from fast memory, the page is chosen from the back of the list.
- **LRU** (Least Recently Used) New pages are added to the front of a list, and pages are removed from the back of the list. Whenever a page is accessed it is moved to the front of the list.
- **LFU** (Least Frequently Used) The accompanying replacement policy for addMFU. Replace the least frequently used page.
- **LFU8** (Least Frequently Used, 8bit) Like LFU, but the *touchCount* is an 8-bit saturating counter.
- **BiLRU** (Bi-Modal Least Recently Used) Like LRU, but 1 out of 128 pages are placed at the back of the replacement list.
- **SCLRU**: (Stream Conscious Least Recently Used) Like LRU, but streaming pages are put at the back of the list when they are added and they are not moved to the front of the list when they are touched.

5.4 Early Results

To quickly test different policies, our first simulations swept each application using combinations of the seven addition policies and six replacement policies. For Lulesh and MiniFE (Figure 5.4) the addMFPRU, addMFU, and addSCF addition policies perform best, with addRAND performing reasonably well for MiniFE.

The BiLRU, LRU, and SCLRU replacement policies have very similar performance. FIFO is slightly worse – about 5-6% slower.

For the two molecular dynamics applications, CoMD and Lammps, the results (Figure 5.5) are similar. The addMFPRU, addMFU, and addSCF addition policies perform best, with addRAND slightly (2-4%) slower. For these applications the choice of replacement policy is even less critical, with less than 1% performance difference between the different policies.

These results show several trends. "Frequency" based addition policies (addMFRPU, addMFU, and addSCF) perform best, though a properly tuned random policy can be reasonably effective. Most striking, as long as the addition policy is reasonable, the replacement policy has a rather limited effect on performance. Even a simple FIFO policy is usually within a few percent of the best policy.

The relative unimportance of the replacement policy is also shown by the next set of experiments in which the threshold for the addT policy was varied for each application (Figure 5.6). Applications tended to perform better with higher thresholds. However, they did evidence diminishing returns and the best threshold appears to differ for each application.

Lastly, we explored the effectiveness of "swap throttling" – not issuing swaps to or from fast memory when the DRAM was busy. While throttling did not have a major impact on most policies, it did have a significant impact on the simple addT (threshold) policy (Figure 5.7). Turning off throttling decreased Lammps performance by 32% and Lulesh by 18%.

5.4.1 Analysis

The primary finding of these experiments is that the addition policy is more important than the replacement policy. Different addition policies, or even different addition policy parameters, can have a dramatic impact on performance, while the replacement policy has a more muted effect. Most likely, this is because bringing pages in which are not frequently used entails a high transfer cost and, unlike a conventional SRAM cache, is not as necessary.

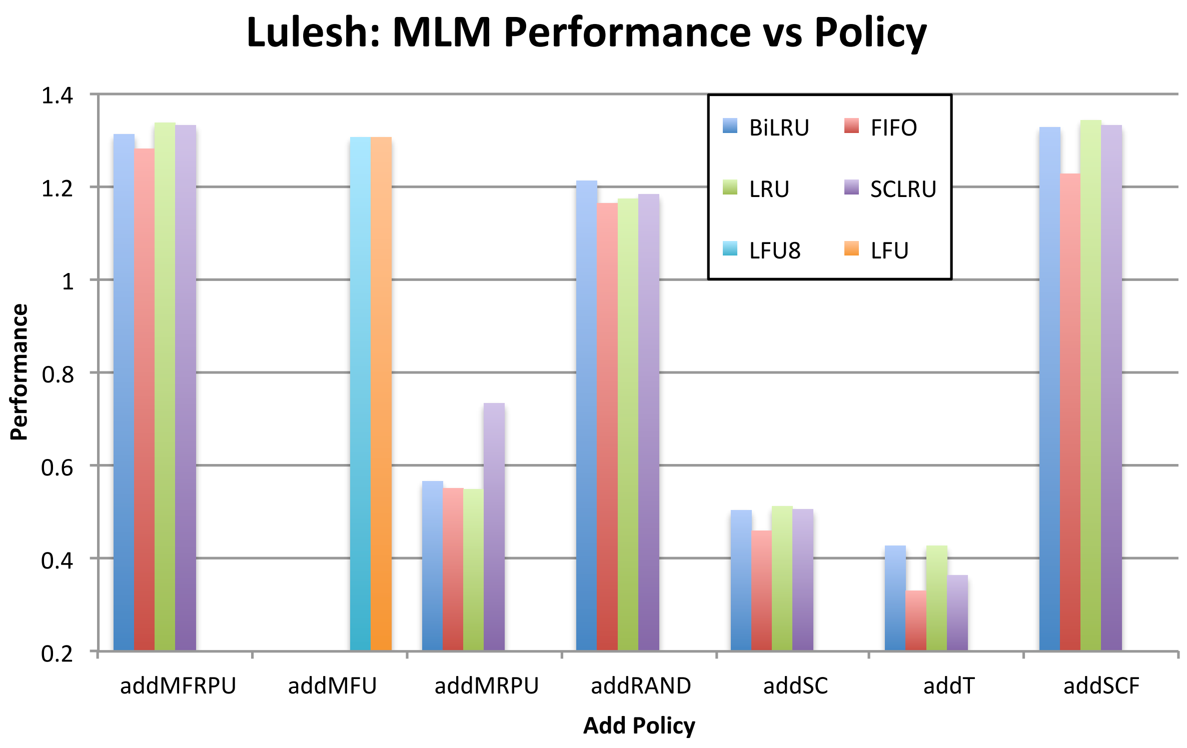
Of the analyzed policies, there are clear "winners." The addMFRPU, addMFU, addRAND, and addSCF all provide good performance over the range of test applications. Each of them present different trade-offs in the amount of meta-data which must be gathered and stored. The addSCF and addMFRPU policies provide better performance, but require collection of *touchCount*s for each page. While addRAND's performance is about 5% lower on average, it does not require metadata collection or streaming analysis.

For these small applications, the "stream aware" policies show promise. Applications which have a number of pages which show streaming behavior (e.g. MiniFE in Figure 5.1) gain a small amount of performance with stream aware policies.

Optimizing policies requires several trade-offs. Simply maximizing he "hit rate" on the fast memory is not sufficient to ensure higher performance. For example the addSC/SCLRU policies actually give a higher hit rate ($\approx$40%) than addSCF/SCLRU ($\approx$12%), even though addSCF/SCLRU provides better overall application performance. However, addSC's lower criteria for addition means that addSC/SCLRU generates 20-70 times more page swaps. These consume considerable bandwidth and hurt performance, even with swap throttling.

5.5 Large Application Results

Guided by the results from smaller applications, we performed simulations on larger applications (Section 2.1) focusing on a constrained set of policies.



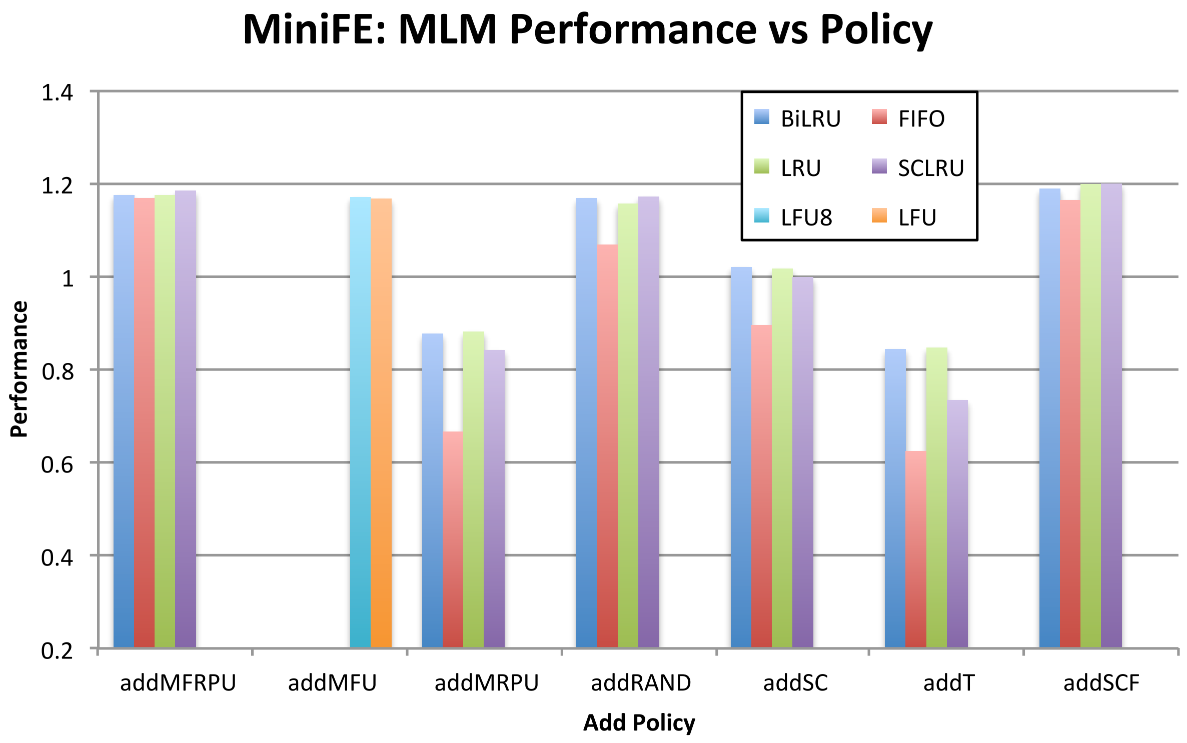
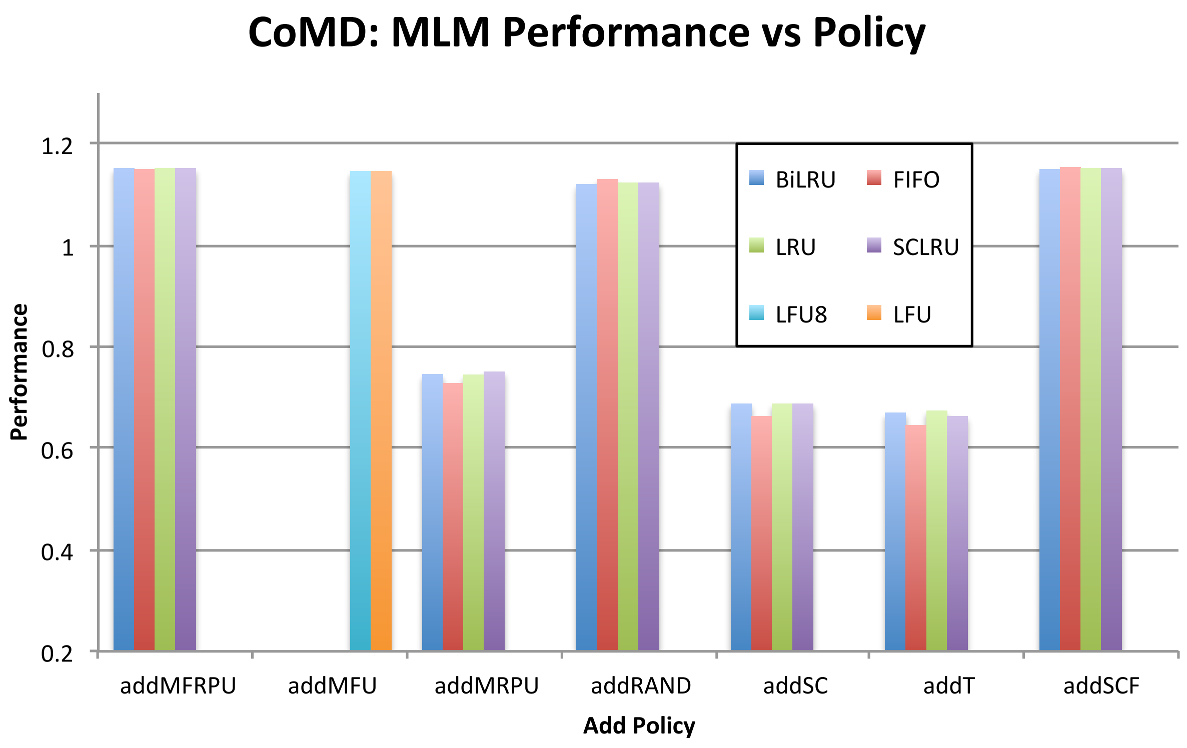


Figure 5.4: Impact of addition and replacement policies with 128 “Fast” pages, normalized to no “Fast” pages.



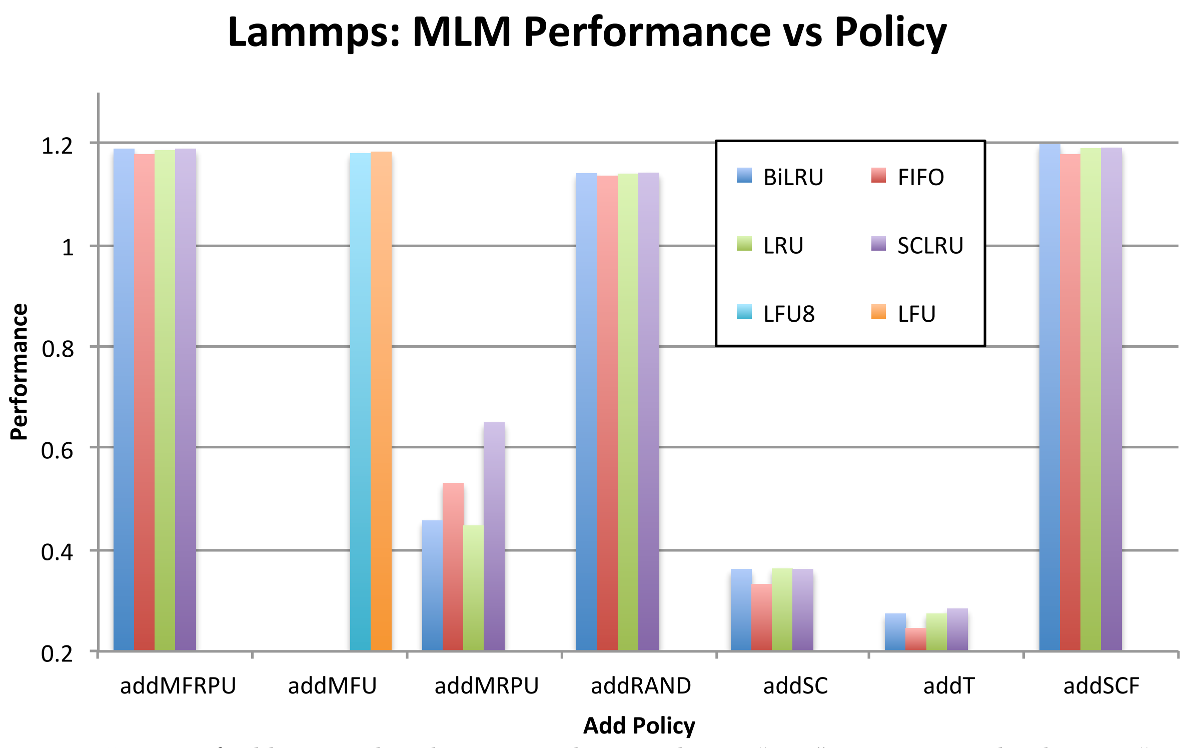


Figure 5.5: Impact of addition and replacement policies with 128 “Fast” pages, normalized to no “Fast” pages.

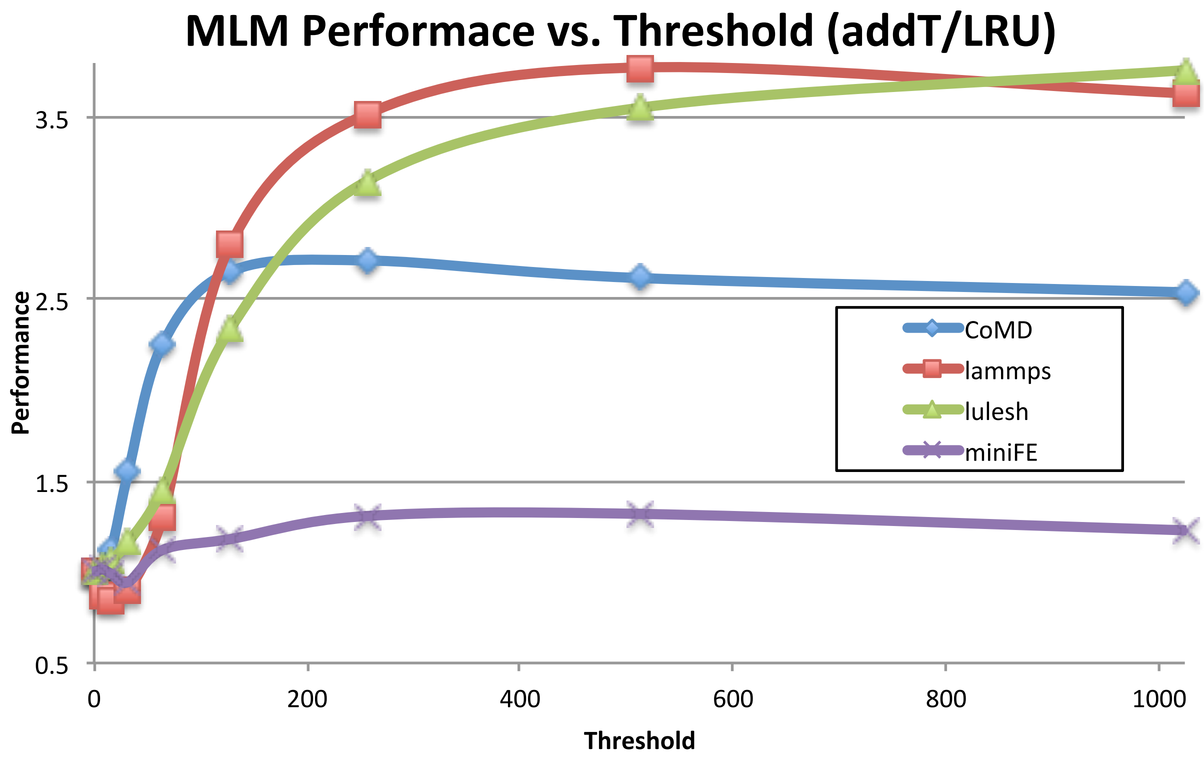


Figure 5.6: Impact of threshold value, normalized to T=0, LRU replacement

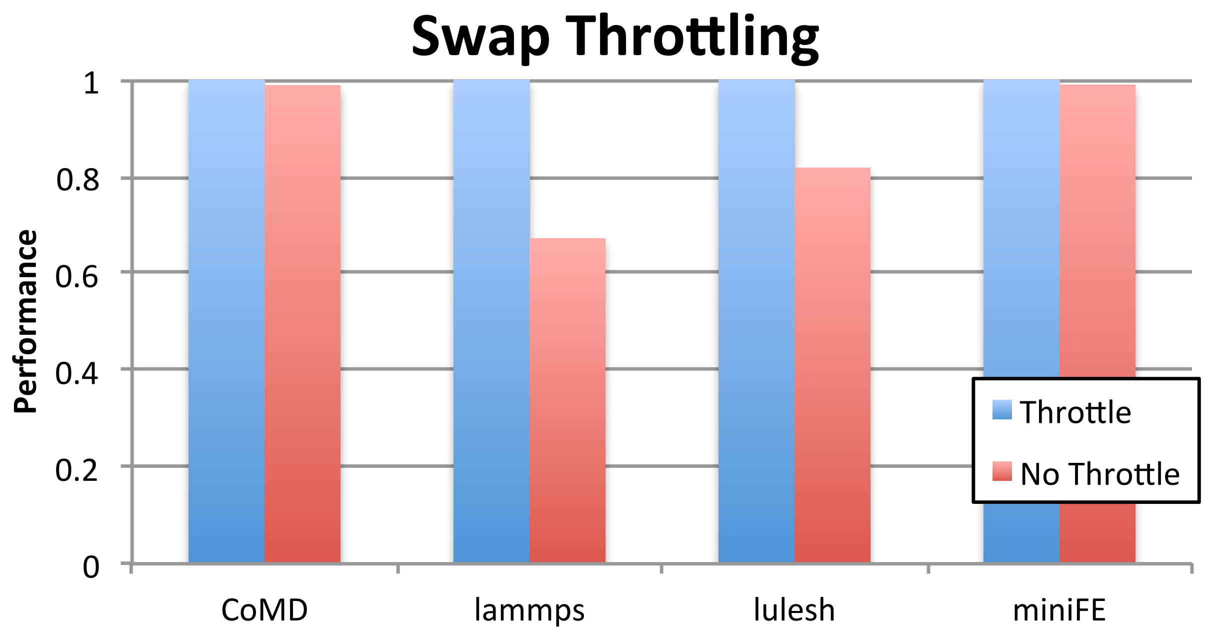


Figure 5.7: Impact of swap throttling (LRU replacement)

5.5.1 Performance Vs. Policy

For replacement policies, we focused on the SCLRU and LRU policies. Across all applications, the LRU policy performed better than SCLRU by 7-10% for smaller numbers of fast pages, but the effect was negligible for larger number of fast pages. Generally, if more then $\frac{1}{8}$ of the memory was "fast" all replacement policies yielded similar performance (Figure 5.8).

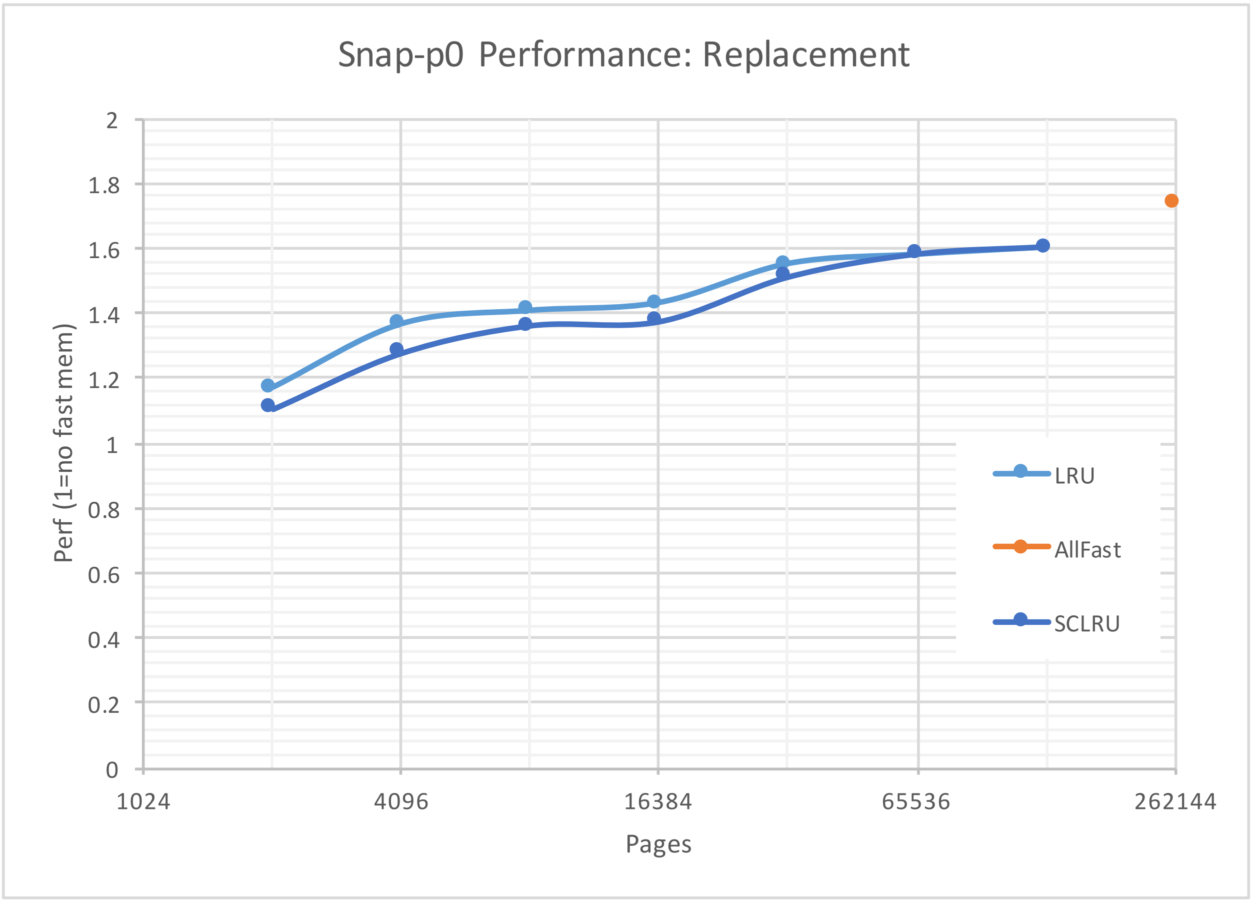


Figure 5.8: SNAP under different replacement policies, normalized to no "Fast" pages. Replacement policy is addMFRPU.

Each application was run with the addSCF, addMFRPU, addMFU, and addRAND addition policies. As expected, addRAND performed poorly, generally giving little improvement over the baseline DDR-only configuration. For all applications the performance of addSCF and addMFRPU were similar. PENNANT (Figure 5.9) performed slightly better with the addMFU policy, though for large numbers of "fast" pages ($\frac{1}{4}$ or more) or small numbers ($\frac{1}{32}$ or less) the difference was negligible. Both SNAP cases (Figure 5.10) saw addSCF/addMFRPU outperform addMFU substantially except for large numbers of "fast" pages. HPCG (Figure 5.11) performance was similar for all policies (except addRAND) with a slight advantage using addSCF.

5.5.2 Fine Tuning

Starting with the basic policies, we examined other parameters to improve performance.

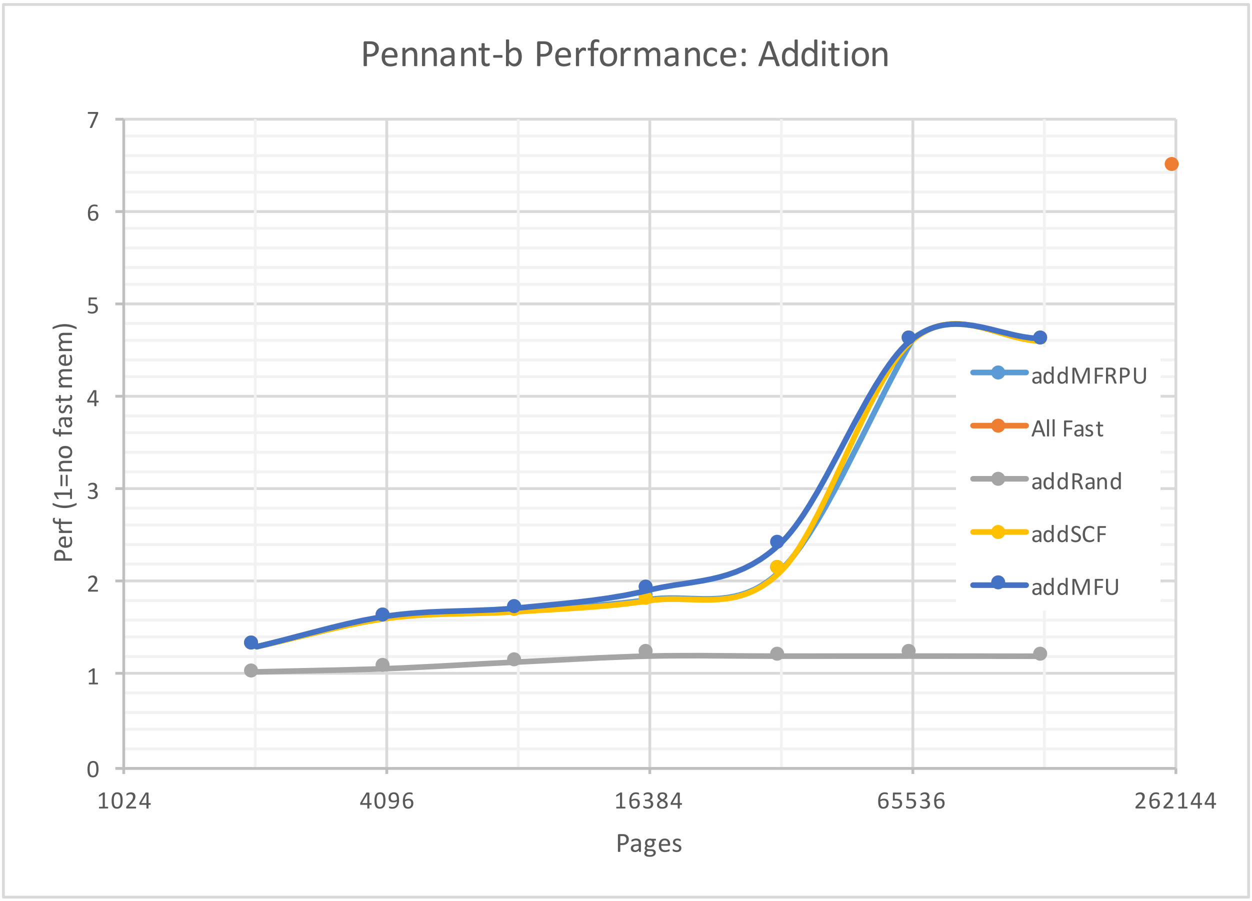


Figure 5.9: PENNANT under different policies, normalized to no “Fast” pages. Replacement policy is LRU.

Like the small applications, the “threshold” value (Figure 5.6) is important. This value is the minimum number of accesses a page must receive in the time quanta before it is considered for inclusion in the “fast” memory. Using the addMFRPU/LRU policies for large applications, it appears that a low Threshold value is gives better performance. This is in contrast to the simple addT/LRU policy for small applications which gives better performance at high thresholds. Presumably, this is because the larger applications are longer running and so a selective (i.e. high) threshold is less important. However, it should be noted that a threshold of zero does not yield the best performance.

Another parameter to explore is page size. This is the size of the block of memory that the policy manages. Our default is 4KB. The 4KB page size works well for these applications, but is not always optimal. For PENNANT (Figure 5.13), 4KB (2^{12}) is the best if 128MB ($\frac{1}{8}$) of the memory is “fast”, however for larger sizes there is little difference. 2KB and 8KB pages have performance which is only slightly lower than 4KB. SNAP (Figure 5.14) works best with pages of 4, 8, or 16KB, depending on the number of “fast” pages. HPCG (Figure 5.15) shows better performance the larger the page size is, however the effect is muted beyond 4KB pages. From these experiments it appears that 4KB pages are a reasonable choice, though larger pages should still yield acceptable performance. This may be important for reducing overhead (Section 5.5.4).

As with smaller applications (Figure 5.7), throttling page swaps between fast and slow memories when the DRAM is busy is important. PENNANT performance was reduced by 20% without DRAM throttling.

5.5.3 Cost & Performance

Automatic management of MLM can improve application performance by 20-380% (Figure 5.16) depending on the application and the amount of "fast" memory. However, MLM comes with added costs. "Fast" memory technologies will almost certainly be more expensive in terms of $ per bit of memory. Additionally, adding extra memory channels or packaging will also impact total system cost.

The exact cost per bit of future memory technologies is difficult to discern, so we examine two scenarios: a cost per bit (relative to DDR) of 1.3× and 5×. Figure 5.17 shows the performance divided by the cost for memory systems with $\frac{1}{128}$th to $\frac{1}{2}$ of their memory comprised of fast memory. In the case where fast memory is only 30% more expensive than DDR, the curves are substantially similar to the raw performance trends. Marginally improved applications (e.g. SNAP) shows diminishing returns after $\frac{1}{8}$th (32K pages). For the 5× case, the trade-off is more pronounced. Adding more than $\frac{1}{16}$th "fast" memory adds more cost than performance to PENNANT and HPCG. For SNAP, the overall performance / cost impact is negative – it would be better to stick with DDR.

5.5.4 Overheads

In addition to the added cost per bit, automatic management also requires additional hardware in the processor or memory controller (Section 5.3). The largest component of this hardware would be the SRAM tables required to store the access count information for each page in fast memory.

Conservatively, we estimate that each page in fast memory requires 32 bytes of SRAM storage. Assuming that SRAM costs 150× per bit as DDR and "fast" memory is 1.3× we can construct a rough cost model which includes overhead (Figure 5.18). This shows that for large "fast" memories and 4K pages, the cost of SRAM is considerable. Memory systems with lots of fast memory would spend almost as much on SRAM as on main memory.

Dividing the performance by the cost (with overhead) makes large fast memories questionable for lower-performing applications (Figure 5.19. For systems with large amounts of fast memory, manual allocation may be better.

The high overheads for automatic management may be reduced through a few optimizations. First, page sizes greater than 4KB would reduce overhead greatly. As shown in Section 5.5.2, many applications continue to perform well with larger page sizes. This could make automatic management of large fast memories more tenable. For example, with 50% of the memory "fast" and 4K pages, SNAP's performance/cost ratio is 83-92% compared to DDR. However, with 8K pages it is 100-109%. For 16K pages, it is 112-124%. Additionally, the choice of 32KB for the per-page meta-information is conservative. It should be possible to use much smaller data structures.

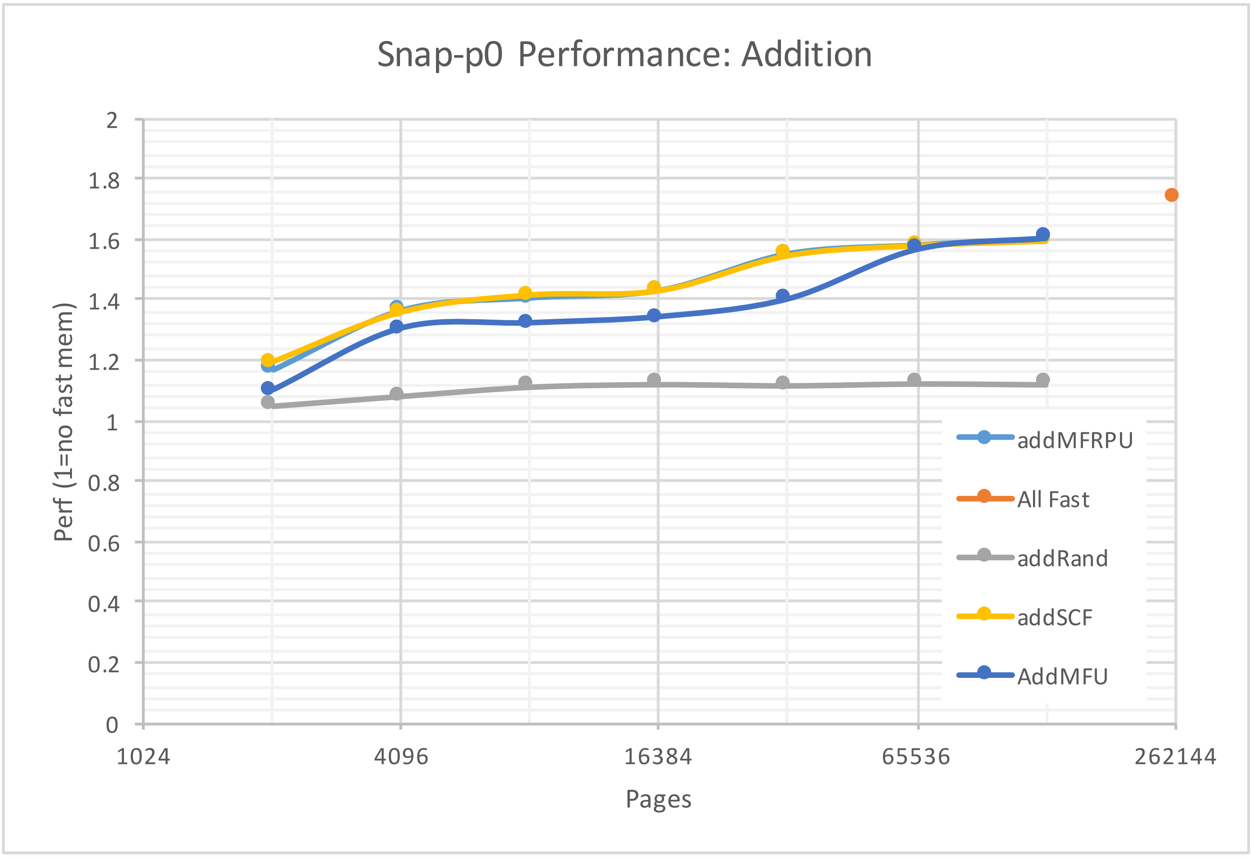
5.5.5 Comparison to Manual Management

We now compare manual and automatic management policies. Note that this comparison comes with an important caveat – because of the simulation infrastructure available we were unable to use exactly the same architecture and simulation models in both studies. For the manual study, we model HMC using GoblinHMC and DRAM using DRAMSim. For the lightweight architecture, we set two of the four memory controllers to be HMC and two to manage DDR DRAM. In contrast, for automatic we use VaultSim for the HMC model and DRAMSim for the conventional DRAM. Each of the four memory controllers manages both types of memory. As such, the manual management's system is likely to have higher performance – a less accurate HMC model, increased memory parallelism, and a lower average network latency. Still, the comparison can tell us the likely trend between the policy types.

With that caveat in mind, we plot the performance of the best manual and the best automatic policy in

Figure 5.20. As expected, for a given portion of "fast" memory, automatic management generally offers higher performance than manual management (Figure 5.20). However, looking at the absolute numbers, manual management of HPCG attains only 38.6% of the performance of automatic management for small sizes of fast memory ($\frac{1}{8}$), though at larger sizes it achieves up to 73.4%. Manually managed SNAP consistently achieves greater than 90% of the performance of automatically managed SNAP. Manual PENNANT ranges from 40-80% of the performance of automatic management. Because HPCG and PENNANT are especially sensitive to available memory parallelism, we conclude that the two policy variants likely perform similarly.

Therefore, the costs, benefits, and viability of each policy becomes central. Automatic mangement requires less programmer effort, but more hardware and a greater capital cost. Manual management on the other hand can require significant programmer investment and so higher labor costs.



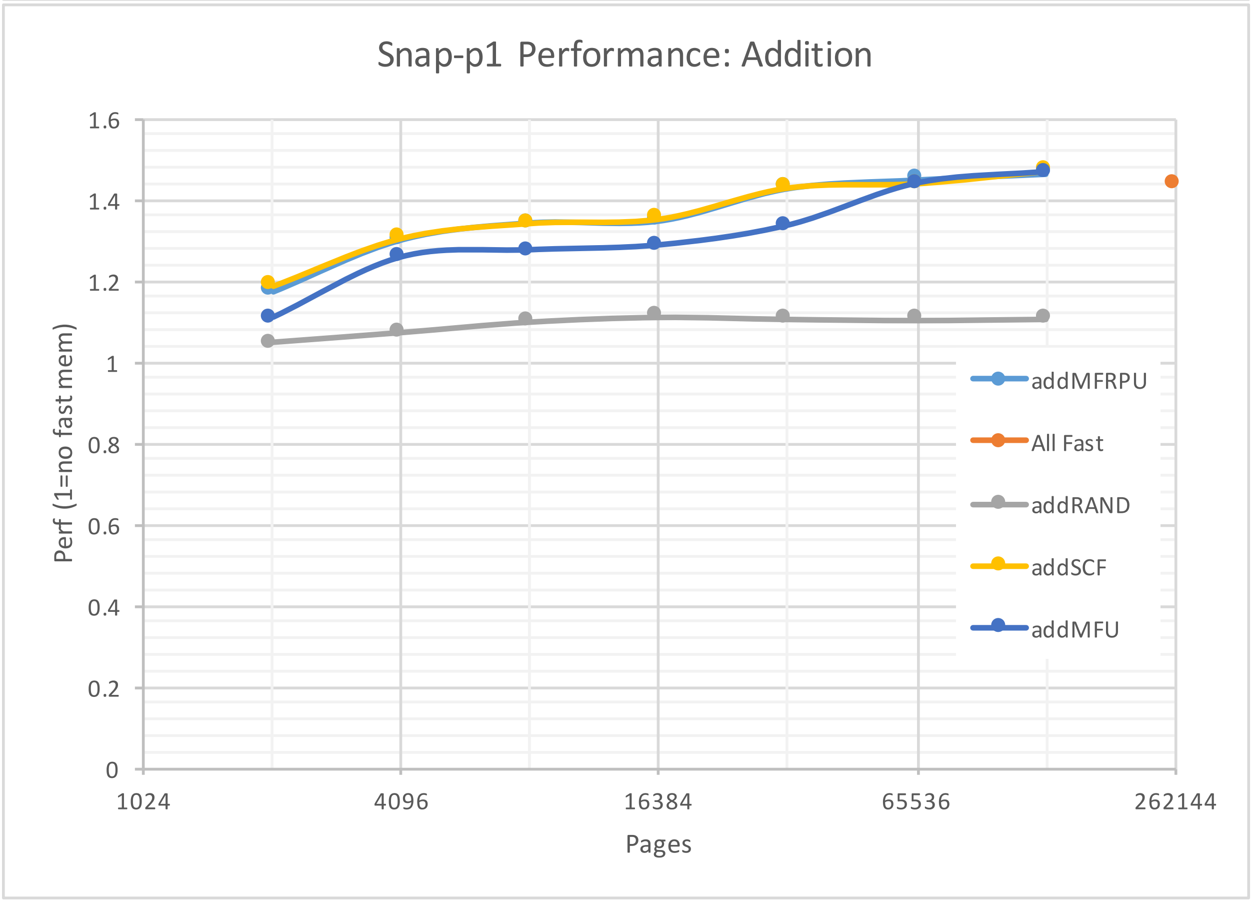


Figure 5.10: SNAP under different policies, normalized to no “Fast” pages. Replacement policy is LRU.

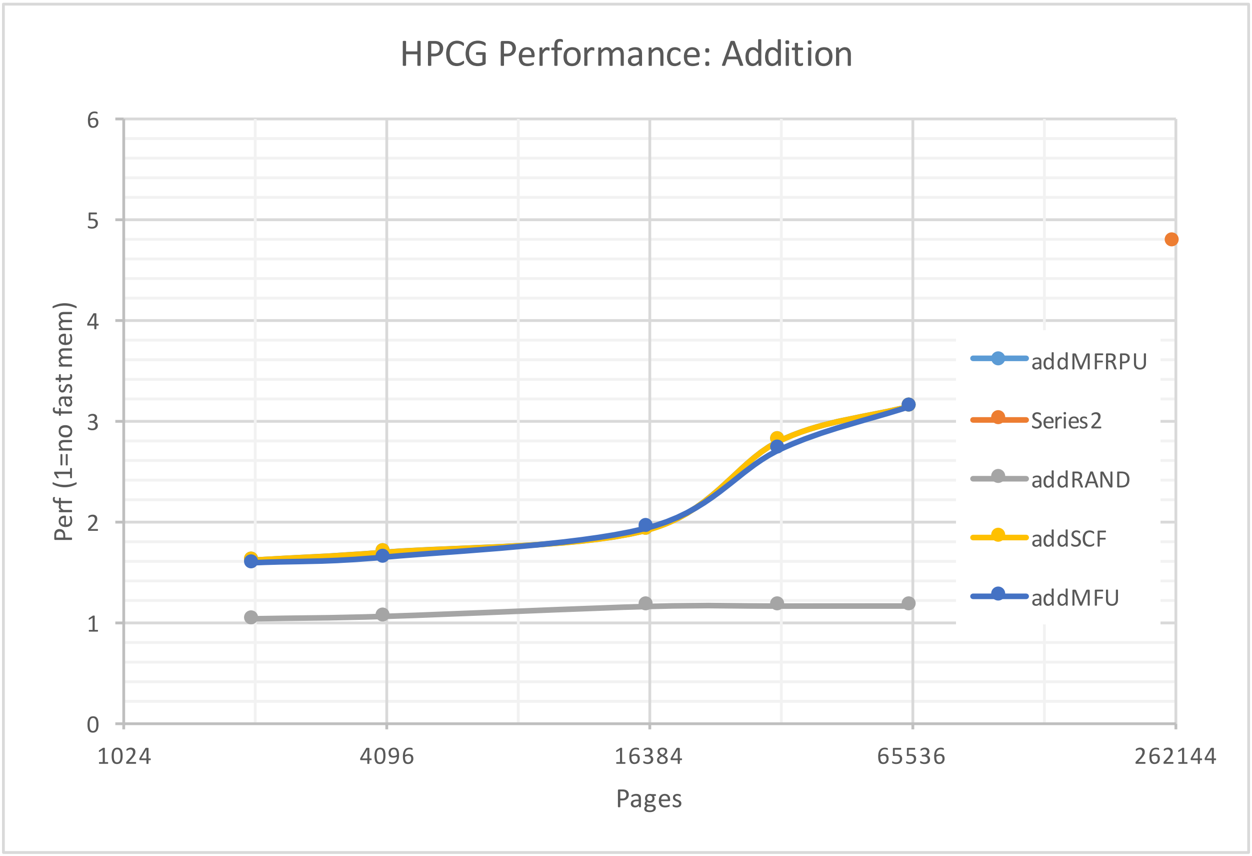


Figure 5.11: HPCG under different policies, normalized to no “Fast” pages. Replacement policy is LRU.

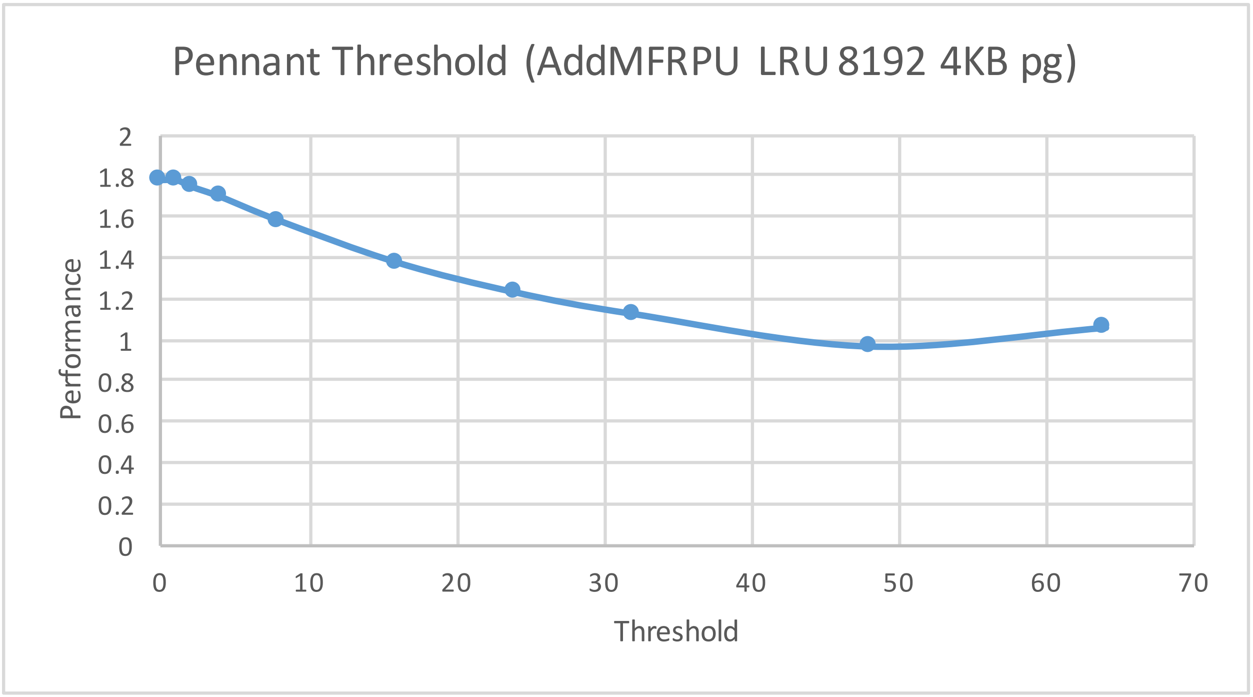


Figure 5.12: Effect of theshold on PENNANT

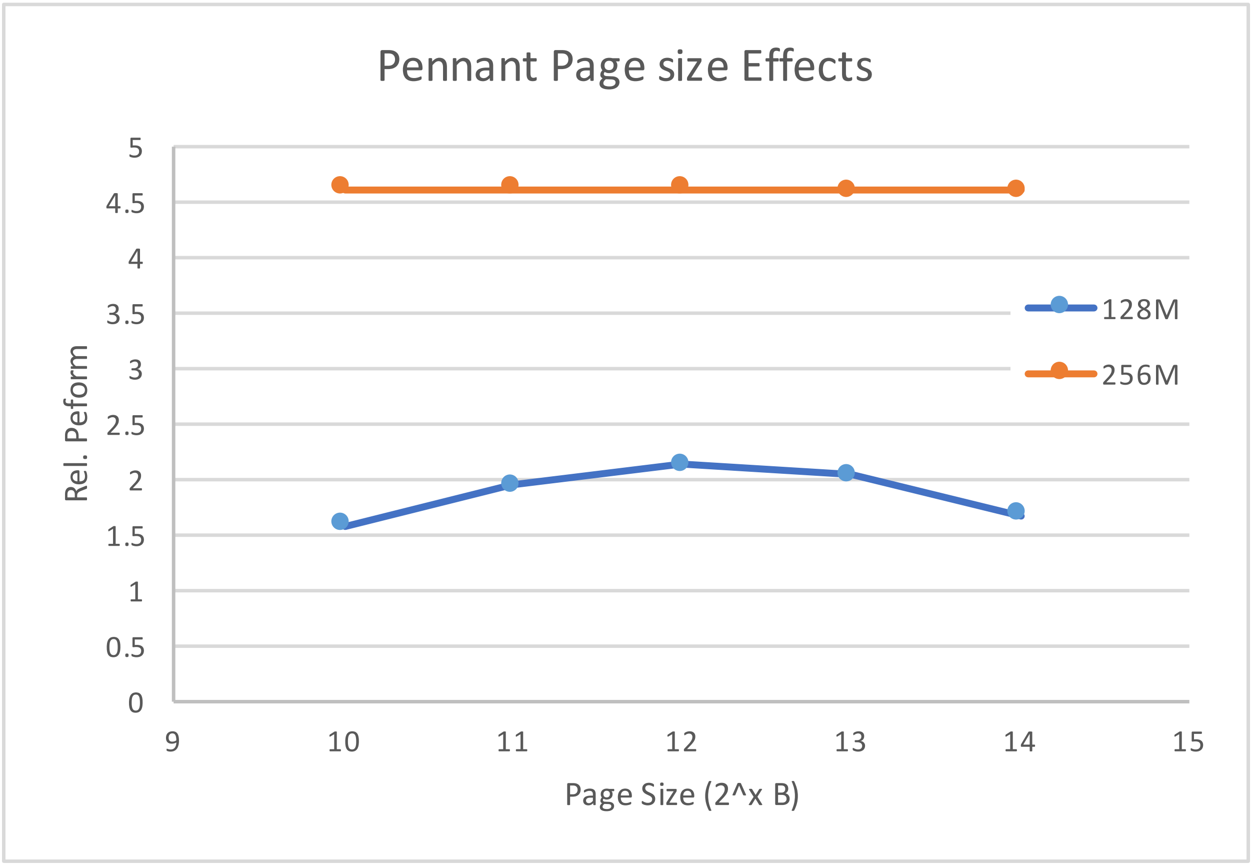
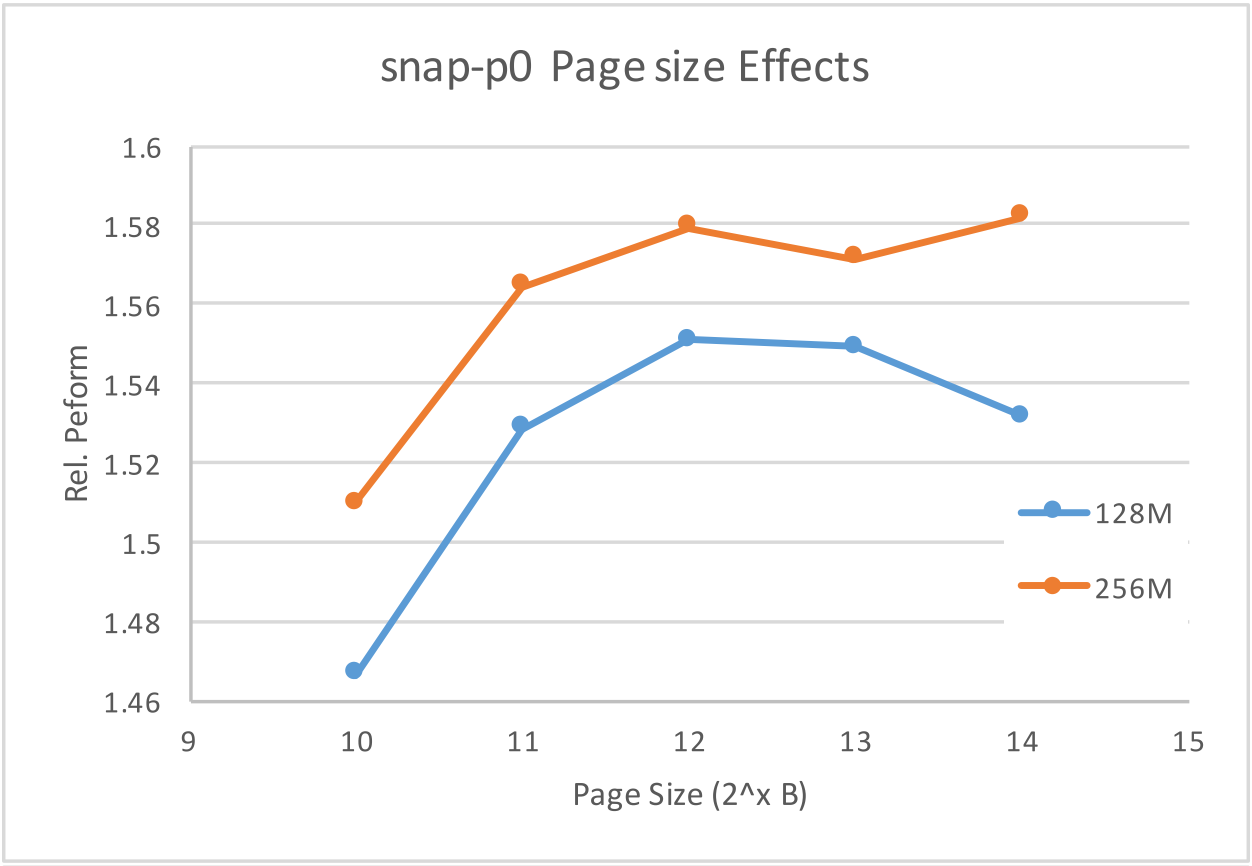


Figure 5.13: Page size effects: PENNANT



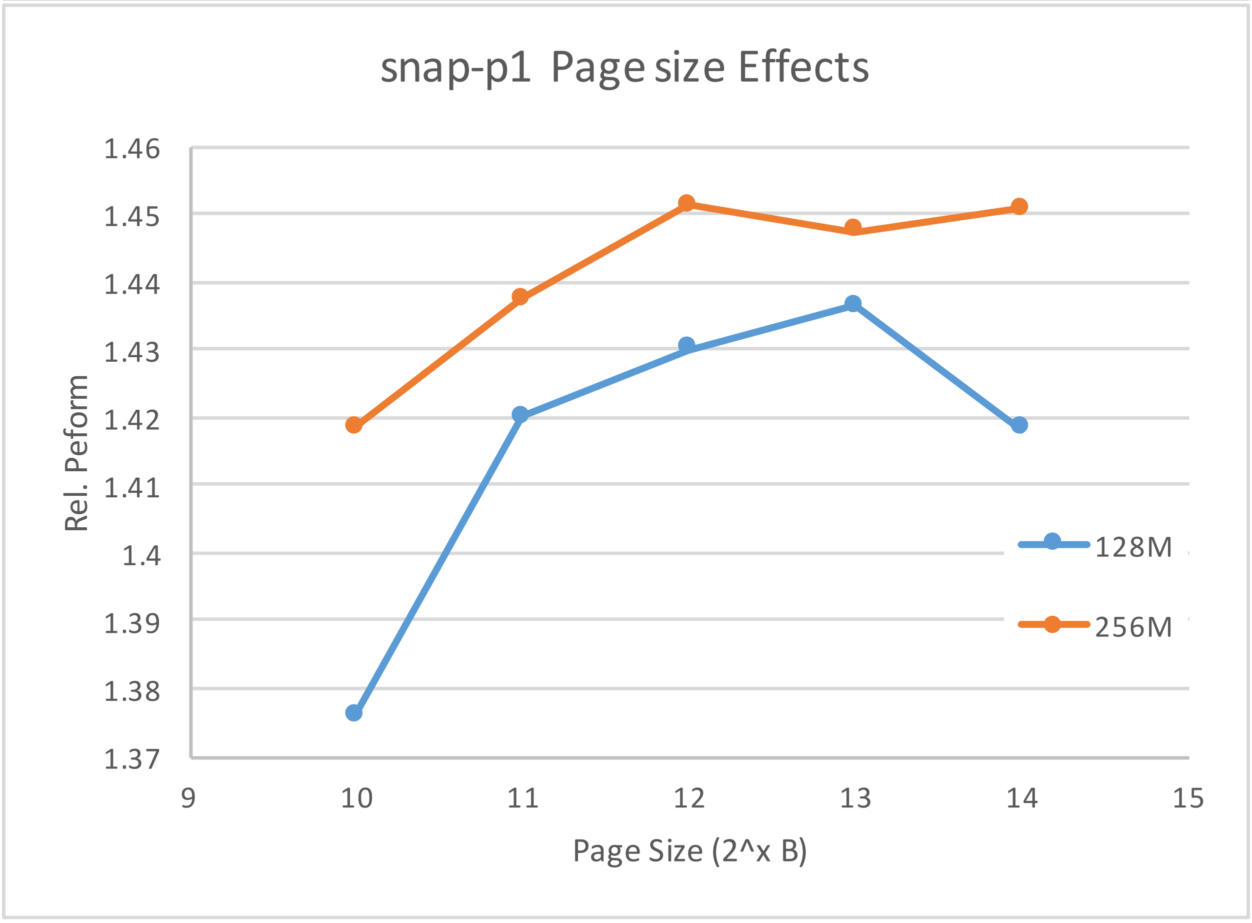


Figure 5.14: Page size effects: SNAP

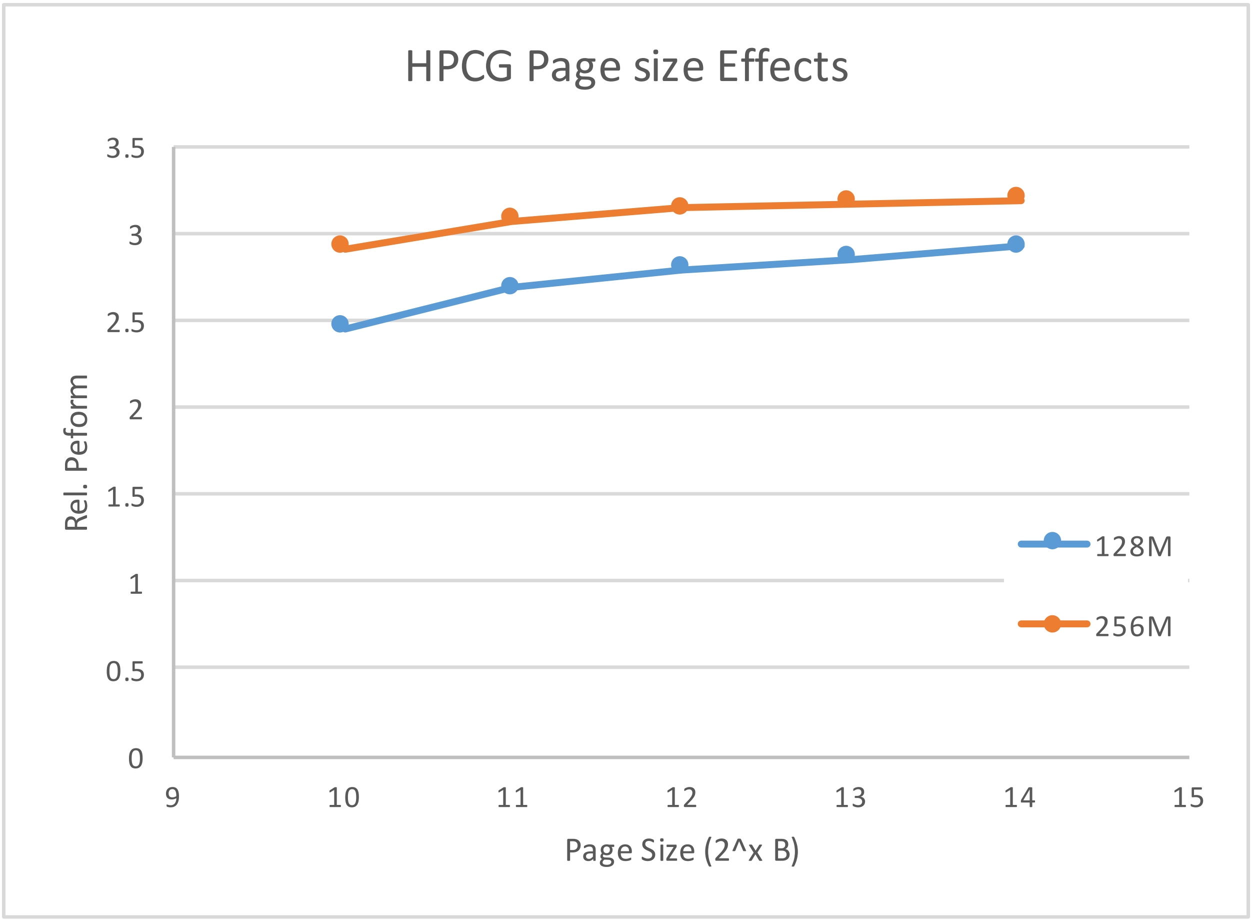


Figure 5.15: Page size effects: HPCG

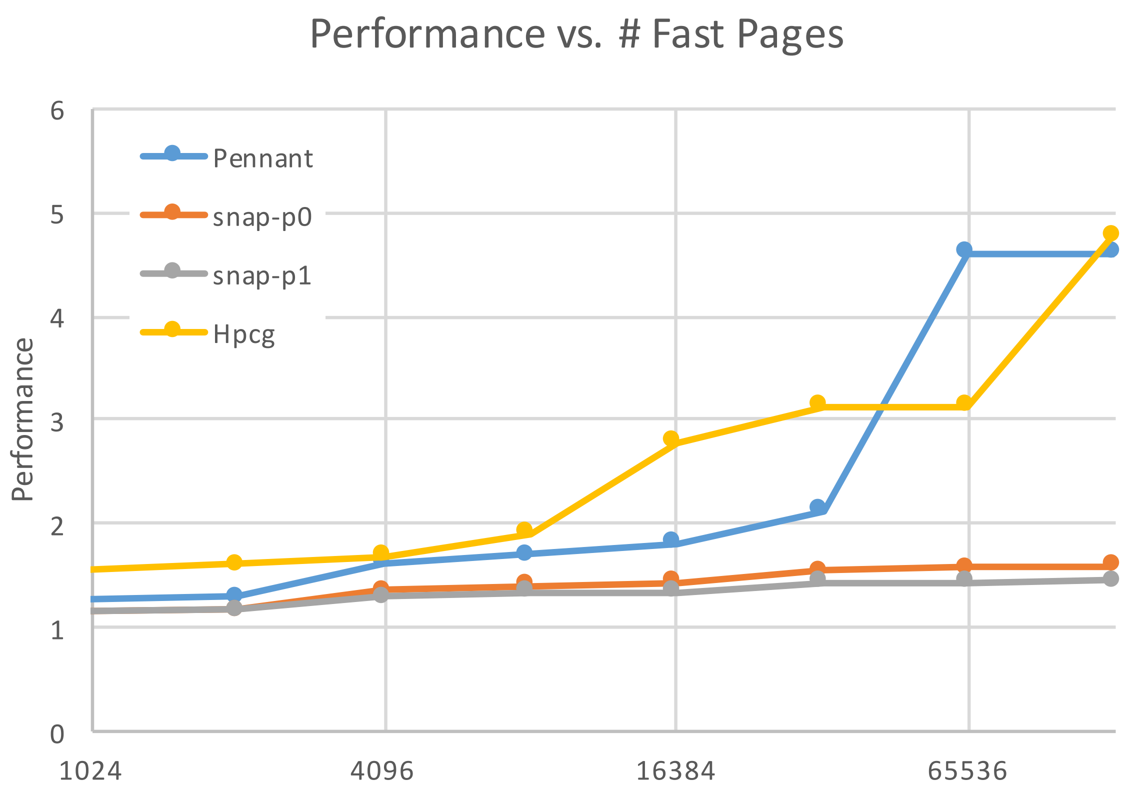


Figure 5.16: Large application performance

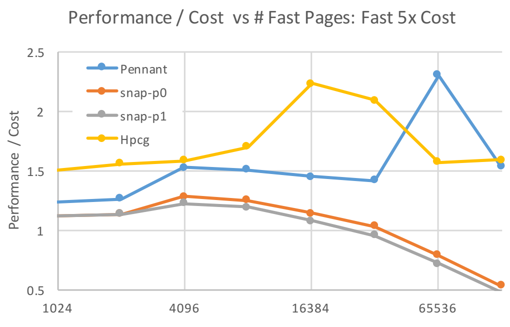
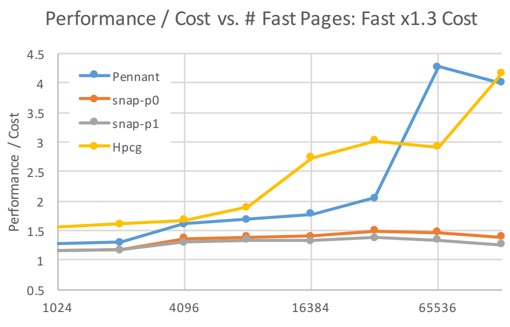


Figure 5.17: Large application performance vs. cost

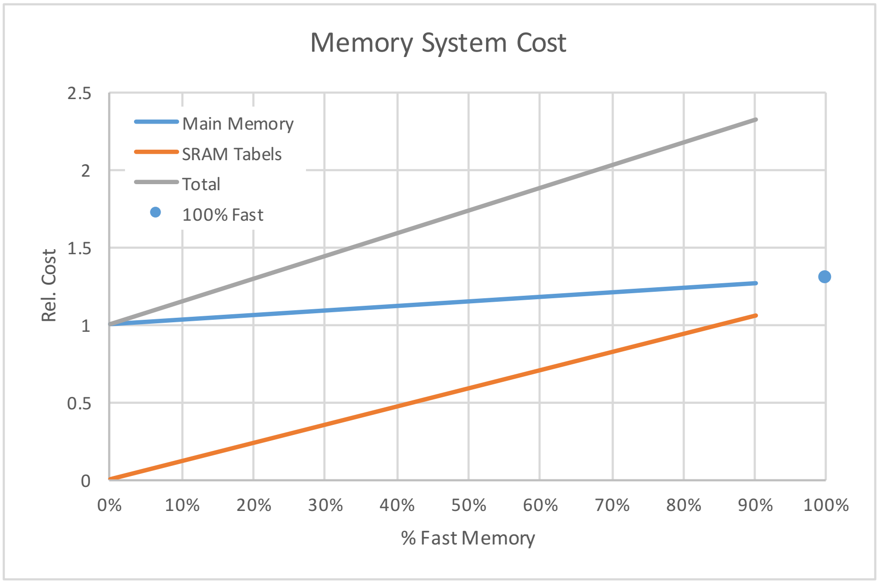


Figure 5.18: Memory system cost, including SRAM tables

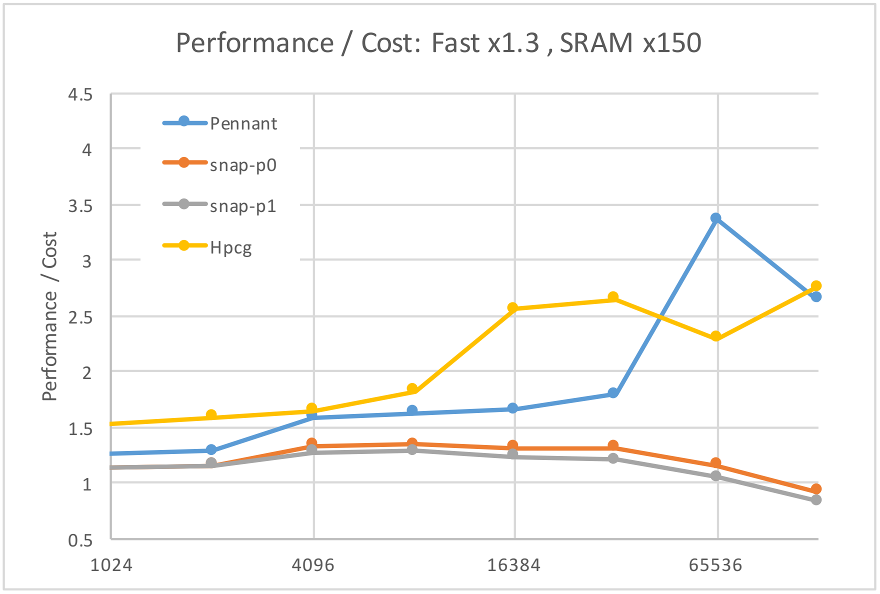


Figure 5.19: Memory system cost / performance

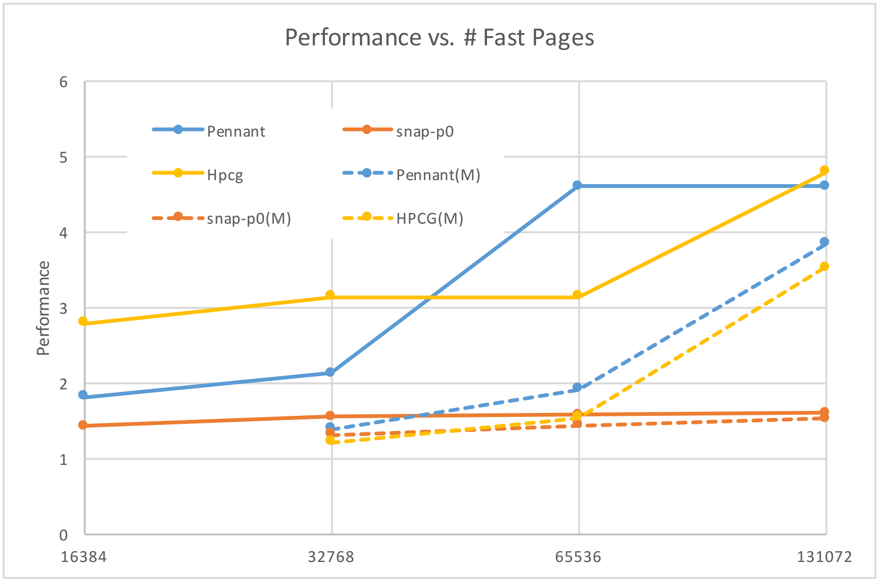


Figure 5.20: Manual vs. automatic comparison

Chapter 6

Conclusions & Future Work

6.1 Conclusions

The emergence of high-bandwidth memory components has the potential to address the decades old problem of the "memory wall" limiting application performance. Components are available today with up to 5X previous bandwidths and into the future even higher improvements will be possible. However, the higher bandwidth comes at higher purchase cost and may be available only in limited capacities. We may therefore be resigned to seeing higher bandwidth memories as part of a larger node with larger, capacities stores also being provided. Such a drastic change in fundamental node architecture raises some interesting potential challenges for our existing high-performance scientific code base.

In this study we have evaluated several aspects to this potential view of the future including application performance in hardware-supported management of resources, operating-system or runtime management and application/programmer directed. We find an array of impacts on application execution and note that a one-size fits all does not appear to stand out from the results in all cases.

The selection of data structures for placement in high-bandwidth memory was evaluated in this study and we find that the selection of such allocations is non-intuitive. Programmers seem to understand which of their data structures will be the "hottest" in access rates but we often find that such intense accesses are absorbed by a well functioning cache hierarchy. It is therefore a second collection of "less-hot" data structures which are the real candidates for high-bandwidth resources. The improvements and new tooling developed within SST for this study show a potential path to identifying such allocations for complex memory systems and architectures that are well beyond those available in the laboratories today.

The assessment of automatic hardware management has also found that one of the most important indicators of system performance is not what page is selected from replacement but which page is selected for addition into a higher-bandwidth resource.

Complex memory hierarchies are clearly challenging our current thinking about application and algorithm design. The potential lost performance of mis-calculating what data structures to place where can be disastrous and therefore it is important that the community has access to scalable and efficient tools and methodologies to trade off design choices. In this milestone we have extended the SST simulation framework to address this challenge. Further, we have validated components within the simulation infrastructure to ensure that our findings are based on accurate predictive mechanisms.

As we look to the future the insights of this study stand out – applications are not comprised entirely of bandwidth bound kernels. Indeed, a variety of bandwidth, latency, and compute boundedness can be seen in our results. For the kernels which are bound by latency, next-generation memory systems will clearly present a significant performance challenge making these kernels a potential bottleneck.

6.2 Future Work

In this study we have focused uniquely on memory systems comprising of two memory pools but future designs may opt to increase this further. We note that this may include the addition of other levels in the memory hierarchy such as non-volatile storage which will be slower than DDR and asymmetric for read/write operations. Alternatively, multiple pools of memory may be added introducing greater NUMA or non-uniform performance effects. SST is well positioned as a tool for future studies in this area.

We also intend to evaluate the energy impacts of multi-level memory system design using SST. Since the introduction of multi-level memory systems can see higher occurrences of data movement it may well be the case that performance is improved but at the cost of much higher power and energy consumption. Our use of SST statistics gathering will form a basis for this work and seek to answer the question of whether such an approach can provide gains in energy efficiency.

6.2.1 Manual Management

The results of manual management techniques indicate a clear need for tools to help identify memory regions or structures for inclusion in different layers of memory. The MemSieve tool is a first cut of such a tool, but requires extension to become useful to the broader audience of application developers.

We hope to extend MemSieve to make it easier to use and faster. Specifically, automating the workflow and simplifying the interface would allow application developers to use the tool. MemSieve can also be made faster. An open area for exploration is the size of a memory footprint required to adequately identify key allocation sites. If it is possible to run an application with a much smaller footprint and still get accurate results for larger footprint runs, the analysis can be performed much more quickly.

Another area for exploration is a combination of automatic management and programmer hints. This may be used to improve performance and/or reduce overhead.

6.2.2 Automatic Management

The current studies show great promise for automatic management of MLM. We hope to expand upon this work in several ways.

More detailed modeling of the memory systems and policy overheads will be pursued. This will include examining the latency sensitivity of the policy unit to determine if parts of the policy decisions can be made by the Operating System or runtime instead of hardware.

There are many chances to optimize policies and policy parameters. It may be possible to auto-tune policy parameters or to dynamically select policies. There are also possibilities for programmer or compiler feedback to guide policies.

The current implementation does not allow accesses to pages that are being transferred between different layers of memory. This can delay a sizable fraction of accesses (up to 8% for CoMD addSC/LRU). A more advanced Policy Dispatcher in the MLM Unit may be able to be less restrictive and allow access to pages which are only partially transfered.

Reducing the hardware overhead of automatic management is critical for many applications and memory sizes. We will explore larger page sizes, reducing the size of meta-data structures, and compression or caching of data structures.

6.3 Recommendations

Finally, we conclude this report with a summary of the recommendations gleaned from the different analyses performed. For architectural and system design our recommendations are:

- Latency should not be sacrificed for bandwidth (Section 3.3)
- Consideration for the management policy must be part of any architectural trade-off study
- Applications vary widely in their ability to use higher bandwidth, but in general a moderate-to-small HMC-to-DRAM ratio should be sufficient
- An HMC/HBM is not a conventional cache (higher bandwidth but not lower latency) so conventional caching policies do not perform well
- For management, we recommend that either a hardware/automatic approach be taken or that the OS/runtime manage MLM with the ability to leverage memory usage information provided by the application ("hints")

For applications we make the following recommendations:

- Use data pools or other techniques to avoid many small calls to malloc
- Very large allocations may be hard to manage efficiently if managing at a malloc granularity
- Try to create program phases (i.e., a particular computation or threaded section) whose memory footprint fits in the fast memory. This will benefit both automatic and manual management policies.
- An API to enable application writers to manage allocations within libraries is needed
- Tools such as MemSieve will be instrumental for enabling applications to get better performance out of MLM systems
- The best allocations for HMC/HBM are those that are accessed frequently but not so frequently they end up in the cache. Identifying such allocations is not always intuitive.

References

[1] Michael A. Bender, Jonathan W. Berry, Simon D. Hammond, K. Scott Hemmert, Samuel McCauley, Branden Moore, Benjamin Moseley, Cynthia A. Phillips, David S. Resnick, and Arun Rodrigues. Two-level main memory co-design: Multi-threaded algorithmic primitives, analysis, and simulation. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 835–846, 2015.

[2] MT Bettencourt. MiniPIC - A Particle-In-Cell (PIC) Code on Unstructured Grids for Next Generation Platforms. In *2015 IEEE International Conference on Plasma Sciences (ICOPS)*, pages 1–1. IEEE, 2015.

[3] Z. Chen, N. Xiao, F. Liu, and Y. Zhao. Ssarc: The short-sighted adaptive replacement cache. In *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, pages 551–556, June 2009.

[4] J. Dongarra, M. Heroux, and P. Luszczek. Hpcg benchmark: A new metric for ranking high performance computing systems. Technical Report UT-EECS-15-736, ECE Department, University of Tennessee, Knoxville, TN, 2015.

[5] C. R. Ferenbaugh. Pennant: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience*, 27(17):4555–4572, 2015.

[6] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving Performance via Mini-Applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[7] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*, chapter Overview of DRAMS, pages 315–341. Morgan Kaufmann, 2008.

[8] Jagan Jayaraj, Arun F. Rodrigues, Simon D. Hammond, and Gwendolyn R. Voskuilen. The potential and perils of multi-level memory. In *Proceedings of the 2015 International Symposium on Memory Systems, MEMSYS 2015, Washington DC, DC, USA, October 5-8, 2015*, pages 191–196, 2015.

[9] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[10] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, LLNL, August 2013.

[11] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[12] M. Pavlovic, N. Puzovic, and A. Ramirez. Data placement in hpc architectures with heterogeneous off-chip memory. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 193–200, Oct 2013.

[13] Steve Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1 – 19, 1995.

[14] Steve Plimpton. LAMMPS Molecular Dynamics Simulator. http://lammps.sandia.gov, March 2016.

[15] A. F. Rodrigues, K. S. Hemmert, B. W. Barret, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, 2011.

[16] ChunYi Su, David Roberts, Edgar A. León, Kirk W. Cameron, Bronis R. de Supinski, Gabriel H. Loh, and Dimitrios S. Nikolopoulos. Hpmc: An energy-aware management system of multi-level memory architectures. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, pages 167–178, New York, NY, USA, 2015. ACM.

[17] Kshitij Sudan, Anirudh Badam, and David Nellans. NAND-Flash: Fast Storage or Slow Memory? Technical report, University of Utah, 2012.

[18] J. Zerr and R. Baker. SNAP: SN (Discrete Ordinates) Application Proxy, Version 1.07. Technical Report LA-CC-13-016, Los Alamos National Laboratory, New Mexico, USA, 2016.

DISTRIBUTION:

1	MS	0845	M.W. Glass, 01545
1	MS	1319	J.A. Ang, 01421
1	MS	1319	K.S. Hemmert, 01422
1	MS	1319	J.H. Laros, 01422
1	MS	1319	S.D. Hammond, 01422
1	MS	1319	G.R. Voskuilen, 01422
1	MS	1319	A.F. Rodrigues, 01422
1	MS	1319	R.J. Hoekstra, 01422
1	MS	1319	M.P. Frank, 01425
1	MS	1319	J. Aidun, 01425
1	MS	1319	M.A. Heroux, 01426
1	MS	1322	K.F. Alvin, 01420
1	MS	0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories